

Flutter Core — Dasar Flutter & Dart

Transkrip referensi — folder sumber `1_flutter_core`

13 bagian · dibuat otomatis

Daftar Isi

1. 1_Introduction
2. 2_basics
3. 3_debugging
4. 4_expense-tracker_interactive-widget-theming
5. 5_expense-tracker_responsive-addaptive-ui
6. 6_todo_internals
7. 7_meals-app_multi-screen-n-navigating-app
8. 8_meals-app_managing-app-wide-state
9. 9_meals-app_adding-annimation
10. 10_shopping-list_form-n-handling-user-input
11. 11_shopping-list_connecting-bacend
12. 12_fav-place-app_using-native-device-feature
13. 13_push-notification

1_Introduction

1. What Is Flutter?

So what is Flutter?

Flutter is actually a combination of two main parts.

For one, it's a UI framework, which is a fancy way of saying that Flutter is a collection of code packages and things you can use in your code to write cross platform apps, cross platform user interfaces.

Because that's one of the main selling points of Flutter.

With Flutter, you can have a single code base and yet generate apps for a broad variety of target platforms, and that's a huge advantage because without a solution like Flutter, you would have to learn different programming languages for different platforms because historically, there wasn't a single shared language that could have been used to target all these platforms.

So that's one of the main selling points of Flutter.

And in this course, you will of course learn how to write that Flutter code so that you can target these different platforms.

But the code alone and this UI framework, which gives you the things you use in your code to build these good looking user interfaces isn't enough.

Instead, Flutter also gives you a collection of tools that you use in combination with your code to then actually convert your code to code

that is able to run on these different target platforms.

Because that code that you write in this single code base does not out of the box work on these different platforms.

Instead, it must be translated to machine code that does work on these different platforms.

So for example, some specific machine code that does work on Android or that does work on iOS, and that translation is also done by Flutter.

To be precise, it's done by these Flutter tools that are part of the overall Flutter solution, if you want to call it like this.

So that you have the things you use in your code, the packages you use in your code to build good looking user interfaces and the tools that you need to then turn that single code base into different code packages for these different target platforms.

Because Flutter is all about building those multi-platform apps based on that one single code base and programming language so that you don't have to learn multiple programming languages in order to build apps for multiple platforms.

With Flutter, it's just one programming language that you have to learn and use.

2. Flutter uses Dart!

So Flutter is all about being able to build apps for different target platforms off a single code base.

However, technically,

Flutter itself is not a programming language.

You are not learning the Flutter programming language.

Instead, Flutter is a framework

for building user interfaces with Dart.

And Dart is the actual programming language.

Now, what's a framework?

Well, a framework is simply a collection of packages

and utility functions and functionalities,

and generally speaking,

things that you can use in your code.

Like for example, some pre-built code

that you can add to your code

to make sure that a button is displayed on the screen.

That's what the Flutter framework gives you.

Now, as mentioned,

Dart is the actual programming language you're learning.

And Dart is a language developed by Google,

which cannot just be used for building mobile

or cross-platform apps with Flutter.

But to be fair, Flutter is the main use case for Dart.

And you use Flutter on top of Dart, so to say,

to build these cross-platform apps.

The Flutter tooling that's all the part

of the overall Flutter solution,

then will translate that Dart code

that uses Flutter features to that machine code

for the different target platforms,

as mentioned in the previous lecture.

But technically,

you as a developer are not just learning Flutter,

but also Dart.

But of course, with this course you will learn both.

You need no prior Dart knowledge.

Instead, you will learn how

to build mobile apps with Flutter.

And along the way, you will also learn Dart,

so that by the end of the course, you know both.

3. One Codebase, Multiple Platforms

So by now you know

that Flutter is a thing you can use

to write code in a single code base

and translate that code to machine code

for different target platforms.

Now, Flutter, as you can see,

offers a broad variety of possible target platforms.

For example, you can build mobile apps for iOS and Android.

And initially, that was all that was supported by Flutter.

And up to this day,

I would argue it's still the primary focus of Flutter.

Or to put it in other words,

these platforms are best supported by Flutter.

And it is the focus of this course.

It always has been.

And with this course,

you will learn how to build mobile apps with Flutter.

Now, what you will learn in this course will still help you

for these other target platforms,

but those won't be the focus of this course.

For example, you could also use Flutter

for building web apps that run in modern browsers

because these web apps use certain features

that are not supported in older browsers.

And you can also use Flutter to build desktop apps

for Windows, macOS, and Linux.

But again, the focus of this course will be on mobile apps.

If you are primarily interested

in one of these other platforms,

this course won't be the right course for you.

Now, what's really important to understand

and know regarding these target platforms

is that you can of course write the code

for all platforms no matter on which machine you are

because it's one shared code base.

It's one code for all these platforms.

But when it comes to then actually producing these

translated code packages for the different target platforms,

you have some restrictions.

Most importantly, you can only test and run iOS

and macOS apps on macOS machines,

Windows apps on Windows machines,

and Linux apps on Linux machines.

So if you want to build an iOS app on Windows,

you can write the code for it because the code is the same.

That's one of the main selling points

of Flutter, as you learned.

But you can't actually generate the app itself

that would then be uploaded to the app stores.

For that, you need a macOS device.

Of course, you can nonetheless write all the code

and then just copy that code to a macOS device,

to a MacBook, for example,

once you are ready to build it for the app store.

But it's worth knowing that on a Windows machine,

you can't build those iOS or macOS apps.

That's important to know and understand.

For Android and web apps, you have no restrictions.

No matter if you're on a macOS, Windows,
or Linux machine, you cannot just write the code,
but you can actually also generate the Android
or web apps themselves based off
that one code base on all these platforms.

4. Flutter Setup - Overview

So now that you know what Flutter is
and what you can do with it,
it's time to get started with it.

And for that, you need to install a couple of tools.

In this video here, I'll give you the high level overview.

In the next video,

I will walk you through all the setup steps needed
on Windows machines.

In the video thereafter, I'll cover Mac OS,
and of course, you can skip the lecture that you don't need.

If you're on a Mac OS device,
you can skip the Windows lecture, and vice versa.

Here in this lecture, as mentioned,
you get the big picture, and then after those next lectures
where I show you the exact steps,
we'll get started creating a new Flutter project,
and previewing our first Flutter app.

So therefore, in order to get there,
let's start diving into those general setup steps.

And here the first step,
as you will see in those next lectures,
is always to download and install the Flutter SDK,
the Flutter Software Development Kit,
and that includes Flutter itself.

So this SDK itself, so this collection of tools
and this Flutter framework, which I mentioned earlier,
but it also means that you will need to install Git,
which is a version control software
that is generally totally independent from Flutter.
It's a software you can use in general, as a developer,
to, for example, create code snapshots
to which you can go back
if you mess something up in your code, for example.
But Flutter, these Flutter tools, which you also install,
actually use this Git software internally,
which is why we'll need to install that.
Again, I will show you the concrete steps,
and we'll show you how to install Git
in those next lectures.
I'm just mentioning it here already.
Now, with that, you would have Flutter installed,
and that would be enough to write the code,
but in order to also build the app for different platforms,
and in order to preview the app
whilst you are working on it,
you'll also need to install some platform-specific tools.
For example, as you will also see in those next lectures,
you will need to install Android Studio
to build and test Android apps,
and you will need to install Xcode for Mac OS
to build and test iOS apps.
As mentioned before, for iOS apps,
you will not be able to install Xcode on Windows and Linux.
You will need a Mac OS device for that,
and that's why I mentioned that you can only preview
and really build those iOS apps if you are on a Mac.

The code is the same,
but these tools can only be installed there.
For Android apps, you don't have these restrictions.
There, you can install and configure Android Studio
no matter on which machine you are.
Now, once you have these platform tools installed,
there's just one last missing step,
and that would be to set up some virtual devices,
some emulators on your system.
For example, an Android emulator
to preview your Flutter project as an Android app,
running locally on a virtual device on your machine.
For iOS, you would create an iOS simulator,
and I will also show you how to do that
in those next lectures.
Using such virtual devices instead of your real phone
is very useful because that means
that you don't have to use your real phone for testing,
and that you instead can use this virtual device,
which can also easily be erased if you need to,
and which is a great tool for testing your Flutter project
as you're working on the code.
That's the theory.
In the next lecture, we'll take a look at the Windows setup,
and thereafter, we'll take a look at the Mac OS setup
before we then thereafter
take a look at how to create projects with Flutter
and how to work with those projects.

5. Windows Setup

In this lecture,
we're going to install Flutter

and all other extra tools
that are required in order
to build mobile apps
with Flutter on Windows.

If you're not on the Windows device,
you can skip this lecture.

Mac OS is covered in the next lecture.

So now, to install Flutter on Windows,
you should visit [Flutter.dev](https://flutter.dev),
the official Flutter website,
and there click on "Get Started."

This takes you
to the official Flutter documentation
and there to the setup page.

And there you can pick the operating system
on which you are
to find detailed instructions
that tell you what to install
on your system
in order to use Flutter on it.

Now here, since this lecture
is about the setup on Windows,
we are of course going to take Windows
as an option,
but it's important to understand
that on this page you're not picking
the platform for which you want to develop,
but the platform on which you develop.

So here you don't pick Mac OS
just because you want to build
a Mac OS app with Flutter,
but instead, you pick Windows

because you are on a Windows system
and you want to use Flutter on that.
Then on this next page,
you find detailed setup instructions,
and I strongly recommend
going through these instructions
in addition to this video,
since these instructions
will always be 100% up to date,
sometimes some extra steps can be added.
So you should always check out this page
in addition to the video.
And on this page, you also get direct links
to all the tools you need to download.
So therefore this is definitely a must-visit page.
Now, when installing Flutter for Windows,
there are some prerequisites.
For example, you must install Git
as mentioned in the previous lecture.
For that, you can simply follow this link
on this page here,
which takes you
to the official Git website,
and there pick the Git version
for your Windows system version
and open that installer,
which is downloaded here.
This will now walk you
through the Git installation process,
and in general,
you can accept all the defaults here
and just walk through these different steps.

But there's one setting you should override,
and that is that you choose

"Use Git and optional Unix tools
from Command Prompt"

if it's not the default for you already.

You should select this option

as also explained

in the official Flutter documentation.

Now with that,

you can continue

through the remaining steps

and confirm all the defaults there,

no changes needed.

And yes, there are quite a bit of steps,

but once you're done,

this will install this Git tool

on your system,

and installing this, as mentioned,

is a prerequisite

to then install and use Flutter.

So once that's done,

you can finish this step

and go back to the official setup instructions,

and now it's time

to really install Flutter itself.

For that, you should download that latest version

using that link

in the official documentation.

And then once this was downloaded,

open this zip file that was fetched here

and extract it,

and in that file,

you will find a Flutter folder.

Now that Flutter folder
must be moved somewhere else,
and it's generally up to you
where you wanna move this,
though the official documentation
recommends picking a path like this.

So here I'll go to my C drive,
create a new folder there,
which I will name SRC,
and paste that Flutter folder
from that zip file into this folder.

So into this SRC folder,
which again can take some time
because it is rather big.

And there are a couple of files
to be moved and copied.

And once copying this extracted Flutter folder
to this source folder here

in my case finished,
you are ready to continue
with the next steps.

By the way, keep these warnings in mind
when picking a location
for your Flutter folder
on your system.

So to continue with the next steps,
you must update your Path,
which in the end means you must make
your operating system
or specifically the Command Prompt
on Windows aware

of these extra Flutter tools
which you did now install.
Because it turns out that this Flutter folder,
which you just copied,
contains a bunch of tools that will be needed later on.
Now to make your system aware of these extra tools,
you should search for "env" in your system search
to add your system environment variables.
Alternatively, you can manually go there
through the Control Panel.
So here on this page,
you should edit your environment variables
and then there your user variables for your user.
Now there you should typically already have a Path variable,
so a variable called Path,
but if you don't have it,
simply create it.
Now I do have it.
So instead of creating it,
I'll select it and click Edit.
And here you now must add a new entry,
a new value to that variable.
And the value that should be added here
should be the path to the bin folder
in that Flutter folder you just copied around.
So in my case, it's this path,
or to be precise, this path,
which then includes slash bin at the end.
So you should copy this path to this bin folder
in that Flutter folder
and insert this as a new value
into this Path environment variable.

And then click Okay.

And Okay.

Now with that done,

you can continue further down to the next step
and run "flutter doctor" in your Command Prompt
to verify that Flutter was installed correctly.

So for this, open your default Command Prompt,
which is pre-installed on your system,
and in there run "flutter doctor."

Now if you get an error
that the Flutter command can't be found,
you should revisit this step
where the Path was updated
because something went wrong there.

It should find the Flutter command
if you did update the Path correctly.

So make sure you did enter the right path
in the right environment variable
if this "flutter doctor" command fails.

Otherwise, this command should work
and should give you some output
that looks something like this.

Now you might still get some errors in there
and that's totally fine,
but it should find the Flutter tool on your system.
It should generally work.

Now, one of the errors we're getting here
is that it can't find the Android toolchain.

So the tools required to develop Android apps
are not installed on this system yet,
and that indeed is the case.

We haven't done that yet.

So that's indeed what we'll do next.

Now that Flutter was installed
and this Flutter command works,
you should move on to the Android setup
to prepare your system
for Android app development with Flutter.

And that all starts with downloading Android Studio,
which is Google's official Android development environment,
which includes a bunch of tools
that will be used under the hood by Flutter.

So on this page, you should download Android Studio,
and for that of course accept all these license agreements
and then start the download.

And this is also a rather big file.

So downloading this can take a couple of seconds
or even minutes.

And once the download finished,
you should open this downloaded Android Studio installer.

So with that installer started,
you can and should accept all these default values,
especially make sure
that the Android Virtual Device will be installed
so that you can start such an Android emulator later on.

I do recommend that you keep the default installation
location to ensure that everything works smoothly later on
and then continue with the remaining steps
and finish that installation process.

Again, this might take a couple of seconds or minutes,
of course.

Now once this finished,
you should also immediately start Android Studio,
and this will open up a configuration wizard

since you are starting this for the first time.

Now in that wizard,

you should choose Custom options here.

Confirm the default location though.

You can pick any color theme of your choice.

We won't use Android Studio for writing code

in this course anyways,

so it doesn't matter too much.

But you should make sure that you do check Android SDK here,

that you do also pick an Android SDK Platform here

and Android Virtual Device.

Now in my case, I can confirm the default folder,

which it would like to use for downloading

and storing these SDK files

since I have a special character in my user folder path.

And hence I'll pick a different path here,

which you of course might not need to do.

And with that, you can continue,

accept these license agreements,

and finish the setup process.

So once done, you can finish this setup wizard

and a new window should open

where you should select SDK Manager

by clicking on More Actions, SDK Manager.

And here you should for one,

make sure that you do check

that latest stable Android version here

under SDK Platforms,

and under SDK Tools,

you should select the SDK Build-Tools

and make sure that you also got

the SDK Platform-Tools selected here.

In addition,
the Android SDK Command-line Tools should also be selected,
and once you did all of that,
you should press Apply
to make sure that these extra tools will be installed.
Again, you will need to confirm some license agreements
and walk through these remaining installation steps,
which again can take a couple of minutes.
Once this is finished,
you have all the software installed
that must be installed
in order to build Android apps with Flutter.
But now you should again go to More Actions here
and go to Virtual Device Manager.
And here, click on Create device.
This wizard allows you to create
a new virtual device blueprint.
And the idea behind the virtual devices simply is
that you have an Android device emulator
running on your system
to conveniently preview your Flutter app
whilst you are working on the code.
So here you can pick any template of your choice
like Pixel 6 in my case, click Next,
download the latest Android version here,
which again can take a short while.
And then once that was done, finish this,
click Next again,
and on this page under Emulated Performance
pick Hardware if possible,
and then click Finish.
With that, you got a virtual device blueprint,

and you can now start the actual emulator
by clicking on this play button here,
and this will start the emulator on your system,
and you can restart this whenever you want
to have such an emulator up and running again.
You can close it when you are done for the day
when you're done writing your Flutter code,
and you can restart it
through that Virtual Device Manager
whenever you need that virtual device again.
Though I will also soon show you an alternative,
even more convenient way of starting that emulator,
but you always must create such a blueprint,
which we did do now.
And with that, we finished all these setup steps.
You can also run your Flutter apps on a real device,
but I will come back to this part here later in the course.
For the moment, we just need this emulator,
which we did now set up as described here
in the official docs.
And with that,
you got everything you need
to build mobile apps with Flutter.
As mentioned multiple times,
we can't build iOS apps on Windows,
which is why we won't add any setup for that here,
but we got everything we need to build Android apps.
Now, what's really important, if you get stuck at any point
or if you're facing any problems,
use that official documentation.
And of course,
you don't need to go

through all these steps again and again
every time you dive back into development,
you only need to do this once.
From now on, you can create new Flutter projects
and work on these projects
without going through all these steps again and again.

6. macOS Setup

So now that you learned how to set up Flutter
and the Android development environment on Windows
in the previous lecture,
you're going to learn how to set up Flutter and Android,
and also the iOS development environment
on macOS in this lecture.

Of course, you can skip this lecture
if you're working on Windows.

Now, to set up Flutter and all the required tools on macOS,
I recommend visiting flutter.dev.

And there you can click on Get started
to be taken to the official documentation,
and there to the setup instructions.

Here, on this installation starting page,
to which you'll also find a link attached to this lecture,
you can choose your operating system,
so not the system you wanna target
but the system you're on.

And here we'll therefore choose macOS.

And once you clicked on that,
you will find detailed setup instructions.

And in general, I recommend going through these instructions
in addition to this video.

Because sometimes, some tiny steps change.

Some links might change, and that's another advantage.

On this page, you get direct links to all the tools
and all the software that must be installed.

And therefore, you should definitely also check out
these written official instructions
in addition to this video.

Now, for macOS development,
on this page, you learn that as a prerequisite,
you need to have Git installed.

I did mention this in an earlier video already.

And indeed, Git will be installed for you automatically
when you install Xcode.

You might also already have it on your system by default.

And therefore, typically,
you don't need to install it separately
since we will install Xcode anyways,
as we need that to build iOS apps.

But you can also install Git separately if you want to.

Also important, at least at the point of time

I am recording this video,
you have a separate setup step that's needed
if you are on a Apple Silicon device,
so a M1 or M2 or whatever macOS device.

And of course, it's possible that in the future,
this step is not required,

which is why you should also check out
these written instructions in addition to this video.

So here I'll just copy this code here, this command,
and then you can open your default system terminal
and execute that command in there.

You might be prompted to enter your password.

And once you did that,

this should install some extra tooling
or walk through that extra setup step which is required
if you are on such an Apple Silicon device.

With that out of the way,
you can move on to installing the Flutter SDK,
so the Flutter tooling and framework itself.
And you should, of course, pick the appropriate version
for the macOS device you're on.

Here, since I'm on a M1 device,
I'll pick this Apple Silicon version
and download this ZIP file.

This ZIP file then will include all these Flutter tools
and the Flutter framework.

Now I'll wait for this download to finish.

And once the download finished,
you can unzip that downloaded package.

And then, with that unzipped folder here,
just go anywhere on your system.

Here, for example, I'm in my root user folder
and created a tools folder there,
though that name is absolutely up to you.

It doesn't have to be named tools.

But in there, I'll create a flutter folder.

And this name is also up to you
and doesn't have to be flutter.

But I'll move all the content,
all the files that were in that ZIP package,
into this flutter folder here.

Again, the exact path and the names
of these folders are up to you,
but you should store all these Flutter files
and all these Flutter tools here

in one central place on your system.

And once you did that,

you must add flutter to your path, as it's called.

You must make your operating system, macOS, aware

of these Flutter tools which you just installed.

You can run this command here in your terminal

to make that terminal temporarily aware of your installation

and make sure that this really points to that path

where you just moved all these files.

But you would have to rerun this command

whenever you restart your system terminal.

Since this is annoying,

you can click on Update your path here,

or choose this here on the right,

and instead update your path permanently

to make your system and your terminal permanently aware

of these Flutter tools you installed.

Now, on this page here,

you find detailed instructions on how to do that.

In the end, you first of all wanna find out

which kind of shell you're using,

for which you can run `echo $SHELL`.

And this should either point you at Zsh or Bash.

If you're seeing Bash, you should update a `.bash_profile`

or `.bashrc` file in your root user folder.

If it's Z shell, as I have it here,

you should update a `.zshrc` file.

If you don't have these files, you can simply create them.

Here, in my case, in my user folder,

where those dot files should be located,

I have such a `.zshrc` file, and I am using this C shell.

So that's therefore the file I should update.

And you can do this with a regular text editor.

Again, if you don't find that file,

you can simply create it,

though it might also just be hidden.

And therefore, you should make sure

that you do show hidden files in your finder.

If you then still don't find it, you can create it.

Here, as mentioned, I do have it.

And I'll edit it with the default text edit app.

And in that file, you then should insert a new entry,

which is also mentioned here in the official docs,

and that should be this entry here.

So here I'll just copy that,

and then insert it here at the end of this file.

And now this entry also must be adjusted

because here you should point to that folder

where you dragged your Flutter files into,

so in my case, tools, flutter,

and then there, this bin folder.

Now, we already see /bin here at the end,

so I just need to replace this path here,

this part between the square brackets

and the square brackets themselves,

with a path to this flutter folder.

Now, one of the easiest ways of getting this path

is to right-click on that flutter folder here

and then choose Get info.

And then here copy this Where part,

paste it in here, and then also add /flutter,

since that name itself wasn't included,

so that overall it should point at your user's folder,

then at the path where you stored these Flutter files,

and then there this /bin folder.

Thereafter, you can and you should save this file.

And once you did that,

you should close and reopen your terminal app.

And then in here, if you run which flutter,

you should see that path to that Flutter tooling

as a output.

You should not get an error that this is an unknown command

or anything like that.

If you do get an error

that this is an unknown command or anything like that,

check that path which you added

to this dot configuration file a couple of seconds ago.

And with that, you get Flutter installed.

You should now be able to also run flutter doctor,

which is a command you can run

to find out whether everything is prepared

for Flutter development on your system.

And this should now not give you an error

that this command is not found.

Instead, this should now do some initial setup work,

possibly download some extra packages

when you run this the first time,

and then, thereafter, it should show you some output

where some steps might still be missing.

For example, here I got no Android toolchain setup,

because I haven't added all these Android tools yet.

But Flutter is installed.

And in my case, I also got Xcode already installed,

though I will still show you how to do that

later in this video.

So you should get some output here.

And with that out of the way,
we can skip this GitHub step here
and instead continue with the platform setup
for the platforms we wanna cover.
Now, here I'll show you both iOS and Android setup.
You can, of course, also skip one of the two
if you only wanna target one specific platform.
But here I'll show you both,
and I'll start with iOS and Xcode.
Now, for that, you need to download
and install the latest version of Xcode.
And the easiest way of doing that
is to use the Mac App Store.
There, if you don't have it installed yet,
you should be able to download and install Xcode,
which can take a while
because it is a quite big tool, as you can tell.
So you should walk through that.
And once you got Xcode installed,
you must run these two commands in your terminal
to basically allow Flutter to work
with Xcode behind the scenes.
So for that, paste them into your macOS terminal.
You again might be prompted to enter your password,
and that's it.
You might not even get a response or some output here.
But you also shouldn't get an error.
Thereafter, also run this command here
to accept all Xcode license agreements.
And here you can simply press Space
to move to the end of this agreement.
And once you're there, which can take some time,

you should type agree in here and hit Enter.

And that's all you need to do for Xcode.

Of course, I also recommend setting up an iOS simulator

so that you can also preview your Flutter project

on an iOS device as you're working on the code.

For that, you can run this command in your terminal

now that Xcode is installed, like this.

And this should now create and boot a new iOS simulator,

a virtual iOS device, a virtual iPhone,

if you wanna call it like this, on your system.

And whilst we're not using it just yet,

we will use simulators, virtual devices like this,

throughout this course to preview our Flutter apps

as we're working on them.

Now, we're going to skip this step

where we create and run a simple Flutter app for now.

Because we're going to do that in the next lecture.

And here I'll also skip the step

where we run this app on a real iOS device,

which you also can do,

because I'll cover this a little bit later in the course.

For the moment, for the next course section even,

we'll just use this simulator.

And therefore, here we can instead move on

to this Android setup part and quit that iOS simulator,

because we don't need it right now.

Now, if you wanna build Android apps on macOS,

you also must install all the tools required for that.

And most importantly, that's Android Studio,

basically the Android equivalent to Xcode.

For that, you can follow this link

to be taken to the official Android Studio website

and download that latest version

of Android Studio from that website.

Again, make sure you pick the appropriate version
for the Apple chip you're on.

Now, this is also a rather big file, a rather big tool,
so this download can take some while.

So here I'll quickly wait for this download to finish
before I then finish this Android setup together with you.

So now here the download finished for me.

And you can therefore open that downloaded file
and finish the installation
by dragging Android Studio into your Applications folder.

Now, copying that will, again, take a couple of seconds
as it's a rather big file.

And once that's done, you should open Android Studio
and walk through this Setup Wizard.

Now, here in this wizard,
you want to make sure that you install the Android SDK,
and here also such an Android API.

And you can keep that default path, which it suggests here.

Click Next, and then make sure you accept
all these license agreements and click Finish.

Now this will download a couple of packages and tools
that are required behind the scenes, under the hood,
to build Android apps, and again,
this can take a couple of seconds or even minutes to finish.

And once this finished, in this window which opens here,
you should click on More Actions and go to the SDK Manager.

And here, in this window,
we have to install some additional software.

Here in this SDK Platforms window,
you should install the latest Android version

at the point of time you're watching this
by simply clicking on this, by checking this.
Under SDK Tools, you should install these Build-Tools here
and also the Android SDK Command-line Tools.
You should make sure that the Android Emulator is installed
and these SDK Platform Tools.
And then click OK and confirm this.
And this will now again prompt you
to accept some license agreements,
and it will then download and install some extra software
that's also needed by Flutter under the hood.
Now, once that's all done, you can confirm this.
And thereafter, you should run flutter doctor again
to confirm that Flutter has everything it needs
for Android development
so that it finds all these tools that you just installed.
For that, in your terminal, run flutter doctor.
And now at this point,
this should give you some green check marks
for Xcode and Android.
And if it does not, as it does for me here,
it might even show you what you have to do.
For example, here I need to run
flutter doctor --android-licenses
to accept some additional license agreements.
So I'll quickly copy this.
And for me, it actually messed up the formatting here.
So I'll fix this to make sure this is in one line
and confirm this with y
and accept these licenses by typing y for each of them.
And with that, if you run flutter doctor again,
you should now also have a green check mark

for Android toolchain

and, if you installed Xcode before, also for Xcode

and, of course, for Flutter itself.

If you don't have that green check mark for Android,

if Flutter somehow can't locate Android on your system,

you should try running this command,

which is also recommended

in the official Flutter documentation.

Well, and with that, the last step

that I wanna go through here

is that I also wanna set up an Android emulator

so that we also got a virtual Android device

for testing our Flutter projects.

For this, you should first of all enable VM acceleration

on your machine, if available.

If it's not available, that's also not the end of the world.

But you should try to enable it if possible,

following the instructions on this page

to which you're linked from the Flutter page.

And with that, go back to Android studio.

And here under More Actions,

open this Virtual Device Manager.

And in there, click Create device

to create a new virtual device blueprint.

Now, here you can choose from a broad variety of templates.

And I'll simply use a Pixel 6 here,

though you can of course use any template of your choice,

and click Next, then select a fitting Android version.

Here I'll simply pick the latest one

at the point of time I'm recording this.

And you might need to download this

when going through this wizard for the first time,

which is what I'm doing here.

Again, as always, this might take some time

since it, again, is a rather big file.

And with that successfully downloaded,

we can continue and mostly keep all the default settings,

though here, for Emulated Performance,

you should actually choose hardware GLES 2.0,

unless this causes any problems later on,

in which case you can switch back to Automatic.

Here I'll choose that, though,

but keep all the other defaults and then click Finish.

And with that, you got that blueprint created.

Now you can click this Play button here

to start this emulator based on that blueprint.

And from now on, you can of course always restart that

if you're diving back into Flutter development.

You don't have to go through all that setup process

over and over again.

And you will get your Android emulator,

which is now booting up here,

which can then be used to preview your Flutter app

on an Android device, as an Android app.

At the moment, we don't have a Flutter project yet.

But that will, of course, soon change.

And now we got all the prerequisites we need

to create Flutter projects and to then run them

as iOS or Android apps on their respective simulators.

Now, what's really important, if you get stuck at any point,

or if you're facing any problems,

use that official documentation.

And of course, you don't need to go through

all these steps again and again

every time you dive back into development.

You only need to do this once.

From now on, you can create new Flutter projects

and work on these projects

without going through all these steps again and again.

7. Project Creation & Setting Up a Code Editor for Flutter Development

So now that we have Flutter

and all the platform tooling set up,

it's time to finally create a Flutter project

and run a first Flutter app on a device.

For that, you should create a new folder

anywhere on your system.

The exact path is up to you.

For example, on the Desktop,

on both Windows and macOS,

this is a fine folder

and there, create a new folder,

for example, called `flutter_projects`,

written like this.

Important, make sure that if you do choose a folder name

that consists of multiple words, as I did here,

that you use an underscore for separating these words,

not a white space, not a dash,

but a underscore.

And once you created such a folder,

you can use that folder

to then create your Flutter projects inside of that folder.

For that, you can simply open that folder in your terminal,

in your Command Prompt on Windows,

or in this case, PowerShell,

which it opened for me here.

On macOS, you simply open it in your default terminal,
and then in your terminal or Command Prompt,
navigate it into this flutter_projects folder.

Both on macOS and Windows,
you can create a new Flutter project
by running flutter create,
and then choosing any name of your choice,
for example, first_app.

Again, if you do choose a name
that consists of multiple words,
don't use white spaces or dashes.

Instead, use an underscore to separate words.

And this command will now create a new project folder
inside of that folder that you created
where a new Flutter app,
a new Flutter project,
will be created in.

So you got a new project folder
with a new Flutter project inside of it.

And the command is exactly the same,
no matter if you are on macOS or Windows.

This will now create a new Flutter project folder,
and it also shows you what you can do thereafter
to run this Flutter app,
but we'll not do this just yet.

Instead, I first want to open this newly created project
in a code editor
because, of course,
building Flutter apps is all about writing code,
and for that, you do need such an editor.

Now, whilst you could use Android Studio
for writing your Flutter code,

I personally recommend Visual Studio Code,
which is an amazing
and totally free, highly-performant code editor
and which can be installed for all main operating systems,
macOS, Windows, Linux.

You can use it on all these operating systems.

And if you visit code.visualstudio.com,
it should automatically select the right version
for your system here.

Otherwise, you can click that arrow
to pick the exact version you do want to install.

This is the editor I will use throughout the course,
and therefore, if you wanna follow along
as smoothly as possible,

I recommend that you also install this editor.

Now, installing it is straightforward.

You simply download that installer here
and walk through it.

And once you install VS Code,

you can open it,

and you should see a screen
that looks something like this.

Now you can go through these walkthrough steps here
to learn more about VS Code.

Attached, you'll also find a short free video

that gets you started with VS Code

and its most important features,

and you can customize its appearance just as needed,
for example, by setting the color theme.

Here, I'm using Dark+,

but you can, of course, use any theme you want.

You can also go to View

and hide and show various sidebars here.

And once you're done,

in the main sidebar,

you should click Open Folder,

and there, select that first_app folder,

which you created a couple of seconds ago,

and open it with VS Code.

And this now opens this project in Visual Studio Code

so that you can conveniently work

with all the files and folders

and all the code in this project.

Now as you see,

this project actually comes with a lot of files

and folders out of the box,

but I will, of course,

walk you through all the important folders

and files in this course.

Now before we dive into any code or folders, though,

there's one extra configuration we should add to VS Code,

one extra thing we should add to VS Code,

and that would be a specific extension

because besides many other amazing features,

one very cool feature of VS Code

is that you can add extensions to it.

And there's one specific extension you should add

by going to the Extensions view,

and that would be the Flutter extension,

which is an official extension

provided by the Flutter team,

which helps with Flutter development

and which is a must-have when using VS Code

for Flutter development.

Now you should install that.

It's totally free to use,

and once you installed that,

you're ready to use VS Code for Flutter development.

8. Running a First Flutter App

So now with VS Code being set up

and this first Flutter project being created

and opened in VS Code, attached to this lecture,

you'll find a main.dart file and you should take that file

and go to that lib folder

in that Flutter project which you created

and which you opened in VS Code

and replace the main.dart file you find in there

with the one you find attached, simply to make sure

that we all have the same starting state.

Now at the beginning of the next course section,

I will walk you through these folders

and files that were created here.

And of course in the next course section you are also going

to learn how to write Flutter

and therefore also dart code from the ground up.

This main.dart file here

of course already has some code in it

so that soon we can see something on the screen.

At the moment, of course,

all this code won't tell you much, though.

That's no problem, though.

You're going to learn all about this code

and much more throughout this course.

So now with that main.dart file being replaced,

we can use this starting code, this demo code,

which I provided to you here
to see a first app running on an emulator.
And for that you should make sure
that you have a virtual device, an Android emulator
or an iOS simulator up and running.
And if that's not the case, a simple way
of running such an emulator is to use VS Code
and then there under View, choose the Command Palette
or use the shortcut, which you see here,
which will open this command window here
where you can type commands
and execute them from inside VS Code.
And for example, if you type Flutter here you'll see
some Flutter-related commands.
You could for example also create a new Flutter project here
with this tool if you wanted to.
But if you then add a bit more context here
and you for example also start typing emulator,
you should see this launch emulator option here.
And if you choose that, a new window should open up
that allows you to pick an emulator.
And here I'll use that Android emulator,
which I created before.
And if you confirm this here, this emulator will be started
for you and that's simply a bit easier
and more convenient than doing that through Android Studio.
And you could of course also start
an iOS simulator from in there.
In this course, I will mostly use the Android emulator
though since that runs on all systems,
whereas an iOS simulator can only be started on Mac OS.
And since not all students have Mac OS,

I will stick to Android for most examples.

Now with such a virtual device up and running,
we can run our Flutter app on it.

It's a simple app, but still this code,
which I provided to you, creates an entire Flutter app
that can be executed on a mobile device.

Now to run this app on a mobile device,
you get a couple of options.

First of all, I recommend that you go to View
and then there under Appearance, you make sure
that you show the status bar,
which is this bar at the bottom.

There you should see the device which will currently
be used for running your Flutter app here.

And if you click on this device, you could also change it
and for example, test it as a Mac OS or Windows app
or anything like this.

This course here is about building mobile apps
with Flutter though, and therefore here I will always use
such a Android emulator or the iOS equivalent.

But here you can make sure
that you will run the app on the correct device.

And once you did make sure
that the correct device was picked,
you can run this app here by going to the main.dart file.

And there you should see a couple of options
above this main thing here,
and here for example, you also have a Run option
and if you press this option here, the app would be built
for the platform you chose here in the status bar
and it would be executed on that device
which you selected in the status bar.

Now alternatively, you can also use the terminal,
which you can also open through VS Code
and then there choose New Terminal
and there run flutter run.

This would also run this app here on the device,
which is currently selected in the status bar.

Now the approach I will use in most cases
is a different one, though.

In VS Code with a dart file selected,
the main.dart file or later, also another dart file,
you can go to the menu bar
and there under Run you find the Start debugging
and Run Without Debugging options.

Now I'll explain what the difference is later in the course.

In most cases in this course I'll use
this Run Without Debugging option, which will also build
and run this Flutter app for the device
that is selected in the status bar.

So here, if you then press Run Without Debugging,
you will see that a new process starts
where this app is being prepared for being executed
on that target device.

This window here also opens up this debug console
in which you could see error messages
if something goes wrong.

You can also close this window, though.

The build process will still continue.

And this little menu here, this little item here,
which also can be moved around,
also opened up in the editor.

This can be used to control the running app.

You can, for example,

stop it by pressing this stop button here.

You can restart it to make sure

that all the changes you might have made in Code

are picked up by using this button.

In addition, this little window here at the bottom might

have opened up, allowing you to start the Flutter DevTools,

which I won't do here, though,

but about which you will learn later in the course.

And now with that app up and running,

if you do go back to your emulator,

you should see an app that looks something like this

running on that virtual device.

Of course, a very simple, not too useful app,

but a first Flutter app running on this

virtual mobile device.

And everything you see here on the screen is in the end

defined in this main.dart file

and is controlled by that main.dart file.

Again, you will of course learn about all that code

in that file throughout the course.

This lecture here is only there

to give you a first glimpse at this code

you will learn about throughout this course

and of course to also show you

how such a Flutter app can be run on such a virtual device.

Now with that app up and running,

let's also make our first little code change

to see how that could be done

and how those changes would be reflected

on that virtual device.

For that back here in the main.dart file,

if you scroll down,

you should at some point find these text things here.

These are so-called text widgets,

but again, you will learn more about widgets

and how to build user interfaces

with code in Flutter throughout this course.

Now in this text widget here, in this first one,

you find this text here, Flutter - The Complete Guide.

And let's say we want to change this code

and we wanna rename this

to Flutter - The Complete Guide Course.

We can simply make that change.

And if you then save that code,

and at the moment it's unsaved, as you can tell,

in VS Code by this dot here

that's showing up next to the file name in this tab.

But once you save that change, for example,

by pressing Ctrl + S or Command + S on Mac OS,

you will see that we had these little windows open up

at the bottom here,

which in the end tell us that the Flutter extension

that we installed here in VS Code noticed those changes

and automatically updated the app

running on the virtual device.

So without us doing anything,

you see that indeed there the UI was updated

and this change can be seen there.

Now sometimes it happens

that changes you make are not picked up,

and in that case you can press this flash icon here

to force Flutter to take another look at your code

and apply these changes or to use this restart icon

and basically restart the app, which will always ensure

that all your saved changes are reflected there.

And these features simply exist
to make the overall Flutter app development process
as convenient as possible since you don't have
to restart the entire emulator or wait for the app
to be rebuilt every time you make a tiny change.

Instead, as you saw, when you make a change,
it's typically reflected in the running app instantly,
which of course is great
and allows you to develop in a very fast pace.

9. Understanding Material Design

So now that we
started this app in the last lecture
and we also had a first glimpse
at some Flutter and Dart code,
I wanna come back to one thing you can see in that code.

And that's this material app thing here.

This thing here is a so-called widget,
as I mentioned in the last lecture,
but you will learn more about widgets
and what widgets are all about later in this course
in the next section to be precise.

But one thing that's noteworthy
about this material app widget here is its name.

Material app.

And it's noteworthy
because it includes a very important word, material.

Because Flutter apps by default use material design,
which is Google's flexible design system.

Now you can learn more about material design on material.io,
which is the official webpage

of the material design system and language.

And in the end, this material design thing
is simply a set of suggestions, rules,
and guidelines established by Google
that help you build beautiful user interfaces.

And Flutter embraces material design
and by using this material app widget here,
which we will use throughout this entire course,
you get a lot of base styles and rules
that ensure that your mobile user interfaces
that are built with Flutter will always look nice.

And of course, that's just a basic app here.

Throughout the course, we will build
way more fancy and advanced apps, of course.

Now, what's really important to understand
about material design
is that even though it is a system built by Google,
it does offer you a lot of customization options
and it's highly extendable.

So as you will also see throughout the course,
it's not the case that all your apps will look the same.

You can customize everything
and you will learn how you can style the apps
to look exactly the way you want them to look.

So it's not a prison,
but instead this material design thing here
is really all about giving you a foundation
on which you can build up on.

So you will see this material app
throughout the entire course,
and in the end, this material app thing will be the reason
for a lot of base styles you get in your app

and will be the foundation on which we build up on
as we apply our own stylings throughout this course.

10. About This Course

And now with all of that out of the way
with this first glimpse at the code here out of the way,
it's now really time to get started
and dive into Flutter and Dart.

And therefore, I wanna conclude this section here
by giving you a brief overview of
what you can expect from this course
and how you can get the most out of this course.

And in the end, this course is structured like a pyramid.

We start with the important dart
and flutter fundamentals with the core features
and the base syntax
and all the core concepts you need to build mobile apps
with flutter and dart.

Of course, we're not going to stop there though.
Instead we're then also going to dive into advanced features
after covering the core features in the first couple
of core sections, and you will learn how
to handle user interaction, how to customize the styling
of your apps, how to build multi-screen apps and much more
before we then dive into even more advanced features.

And you learn how you can animate apps,
how you can connect the backend,
how you can use native device features like a camera in your
Flutter app and much more.

Now, what's really important to understand is
that we really do start at the basics though.

You don't need any prior flutter or dart knowledge.

You don't even need any programming experience.

And instead starting with the next section,
you will learn about all these basics step by step,
slowly building up on each other so that by the end
of the course, you will be able
to build your own flutter apps no matter if they're super
simple or slightly more advanced.

Now of course, I also want to ensure
that you get the most out of this course though,
and of course, that's a video on the mount course.

So it all starts with watching those videos,
but definitely do it at your pace.

Use the video player controls to slow me down if I'm going
to fast or speed me up if I'm going to slow for you.

Feel free to skip lectures if you already have some dart
or flutter experience.

Really take advantage
of this on demand nature of this course.

This of course also means that you can
and should repeat videos or entire sections as needed.

If a certain concept isn't entirely clear the first time,
definitely feel free to watch a video again.

It's a on-demand course, take advantage of it,
but don't just use the videos.

Instead, I do recommend that you also do code along
and that you practice what you learned.

Now for that, I will sometimes prompt you
to try something on your own,
but definitely also do it when I'm not telling
you to practice.

Instead, feel free to pause those videos when we're about
to add a new feature and try adding it on your own.

Also, use what you learned
and build your own demo projects outside of this course.
This course gives you the foundation you need
to use it on your own, in your own projects
because ultimately that's
how you become really good in something
and flutter is no exception.
But if you are following along with this course,
also uses the code attachments.
The code snapshots I'm providing to you as part
of the next course lecture,
so the lecture after this one, to compare your code to mine
because if you fail to replicate
what I'm showing in the videos
or if you're facing some error
that's not occurring in the videos,
you can use these code snapshots to compare your code
to mine and find and fix this error.
That's why I'm providing them to you.
So definitely take a look at that next lecture
and use those code snapshots
as you practice what you learned.
You will also find a link
to all the core slides in that next lecture.
Last but not least, also help each other.
This is an online course,
but there is a community behind it.
There is the q and a section, which you can use
to ask if you're getting stuck
and you're not able to continue on your own.
But don't just ask there, also help others, help each other
and also answer in that q and a section

because ultimately if you answer a question
by some other student, you are forced
to think about the problem
and you are forced to come up with a solution
and this is a great practice opportunity.
In addition, this course also gives you access
to a Discord community, which you can join
for free if you want to
to find more amazing developers there so that you don't have
to learn flutter on your own.

2_basics

1. Module Introduction

In this course section,
we're going to build our first basic app.
An app which allows us to roll a dice
and generate random values that are then reflected
by an appropriate dice image in our app.
And whilst the app, of course,
is still fairly basic and not too fancy,
it's an amazing app for getting started
with Flutter and Dart
since it will allow us to write quite a bit of code
and dive into many crucial Dart and Flutter features,
which you will need for the rest of this course
and for basically all Flutter apps you're about to build.
Because this course section is about exploring those core
Flutter and Dart fundamentals, which you will always need.
It is about laying that solid foundation,
diving into the core Flutter and Dart features and syntax.
It is about making sure that you understand
how you can write Flutter and Dart code.
And this section is also about understanding
that in the end, when building apps with Flutter,
it's all about combining widgets,
as you will learn in this section.
And of course, you are going to learn what widgets are,
how you can use them,
and how you can build this app here
step by step throughout this section.

2. Analyzing A New Flutter Project

Now, before writing any Flutter or Dart code,
I wanna walk you through the important folders and files
that were created when you created a new Flutter project,
and for this section,
you can either create a new Flutter project
anywhere on your hard drive in the folder
where you wanna manage all your Flutter projects,
for example, or you use that Flutter project
which we created in the previous course section.
Either way, you should make sure
that you use the main.dart file,
which you find attached to this lecture,
to replace your main.dart file in the lib folder with,
so the main.dart file in the lib folder
in that project which you opened,
and which you now have open,
should be that main.dart file you find attached.
And with that, you should have a new Flutter project,
and the exact number of folders could change over time,
but you should see a bunch of folders in there,
and as you might have guessed already, the lib folder,
for example, is a rather important folder,
because it's that lib folder
that contains your actual Dart code files,
so the files in which you will store the Dart
and Flutter code that makes up your application.
At the moment in there, we got only one file,
but as you will see throughout this section
and the entire course,
you can add multiple files in that folder,
and you can then also combine these files

to split your app code across multiple folders,
which you simply typically do
to keep your individual code files manageable and readable.

So at the moment, it's one file,
and what's of course important about this file
is that it ends with `.dart`.

As mentioned in the first course section,
Dart is the programming language that's used by Flutter
to build apps with Flutter,
and your code files must end with `.dart`, therefore,
also to tell Visual Studio Code,
or whichever code editor you are using,
that the content in that file is Dart code,
which then allows the code editor
to give you proper support and code formatting
and highlighting of certain keywords,
as you can see here on the right,
because that's one of the main ideas
behind using a code editor,
that you get some syntax highlighting, as it's called,
so that certain keywords are highlighted,
that it helps you catch typos
and do many other things for you,
and in order for the editor to properly support you,
it, of course, must know in which language
you are writing your code, and that file extension
does help the code editor with that.

So this `main.dart` file in the `lib` folder is important,
and the `lib` folder therefore is important,
but it, of course, is not the only file
and folder you have in that project.
Instead, you should also see an `Android` folder,

you should see a build folder,
you should see a Windows folder, a web folder,
and also a couple of other folders as you can see here.

Now, these folders that carry platform names,
like Android or MacOS or Web or Windows,
simply contain platform-specific files.

Typically, you don't need to do anything
with the files in those folders.

Instead, Flutter will change
and configure those files for you when needed,
so you don't need to do anything with the files
and folders in those platform folders.

They will be managed by Flutter,
but they still are important,
because they are used under the hood
when you run your Flutter project as an app
on an Android or iOS simulator, and of course, also later,
when you prepare your app for deployment,
when you prepare it to be published
to the Android store or the Apple App Store.

So whenever you wanna publish an app to a specific platform,
the files and folders in the respective platform folder
will become important.

The build folder is also important,
but also a folder in which you won't work.

It only contains temporary files
and output files generated by Flutter
as it builds your app for different target platforms,
so you shouldn't change anything in there,
but it is an important folder
that's also managed automatically for you by Flutter.

Now, in the test folder, you also find Dart files,

but unlike the lib folder, the test folder is not a folder where you will write your main application code.

Instead, in there, you could write code that tests your main application code.

You could define automated tests, which can be very useful for catching errors early and for making sure that you don't have to test everything manually.

Now, testing, automated testing, is an advanced topic and not something we will cover here in this course, but you can still keep this folder around, even though we won't work in it.

And then you also got many more files and all the more folders up there.

Now, these folders starting with a dot simply hold some extra configuration.

For example, the .idea folder contains some extra configuration for Android Studio, Dart tool contains some extra configuration for some, guess what, Dart tooling, and you don't need to tweak those configurations, typically, so you can also ignore these folders here, and then when it comes to all these files down there, we got more configuration files.

A gitignore is a file that's used by Git, this version control software, which you can use for managing code snapshots and going back to snapshots if you messed up, for example, but you don't have to use Git when using Flutter, so you therefore can ignore that file.

You also shouldn't delete it, though.

Now, .metadata is actually a file

that is managed automatically by Flutter
to keep track of some internal information
and metadata about your project,
so you should not delete that file,
but you also shouldn't edit it.

The `analysis_options.yaml` file configures some Flutter
and Dart tooling that's being used by your code editor
to show you warnings and errors in your code
before you even run the app,
which is amazing for catching errors early,
and typically, the default settings here should suffice
and should give you everything you need,
but once you're a bit more advanced,
you could definitely dive into this file
to tweak those settings to your likings so that,
for example, a certain style of writing your code
is enforced so that you get warnings
if you accidentally use another style or anything like that.

This is advanced, though,
and something you can therefore ignore initially.

Now, this `first_app.iml` file is another file
that's managed by Flutter for you,
and you can therefore ignore it.

It contains more metadata and information
for building your Flutter app for different platforms.

The `pubspec.lock` file is another managed file
which you should ignore,
but it is connected to this `pubspec.yaml` file,
which is an important file that you may edit
and that we will edit at certain points in this course,
because this `pubspec.yaml` file,
for example, allows you to add third-party packages

to your Flutter project,
so if you wanna add some feature to your app
which you don't want to build on your own
because it's too cumbersome or complicated,
but it's also not built into Flutter or Dart,
you could bring in a third-party package,
and you will see some examples for that
throughout the course, and you would add those packages
and refer to them inside of this pubspec.yaml file.
So it's not a file you need to edit right now,
but we will edit this file from time to time,
for example, also in this section already,
to include images into our Flutter project.
But again, you will see how that works
once it's time for that.
At the moment, you can ignore that file.
Well, and the readme file simply contains
some general information
about this project that was created, a short description,
and some resources you can visit
to dive deeper into Flutter,
which you, of course, don't need,
because you have this course.
And with that, you now have a first idea
of what all these folders and files do.
Of course, you will see many of these files
and many settings in action throughout the course
as we need them.

3. From Dart To Machine Code

So now that you've got a first overview
of these folders and files that were created

as part of this new Flutter project,
I wanna take a closer look at this main.dart file
or to be precise, what happens to that file
when you run your Flutter app on an emulator
or on any device.

As you can see, this main.dart file contains some code,
which most likely looks quite cryptic to you at this point
since we haven't learned about this code yet.

But of course, we will learn about writing this code
and what this code does throughout this course section.

But you can of course, see
that there is some code in that file.

And I can tell you that this code
like this would not be understood by Android
or iOS devices.

So what's not happening
when you are running your Flutter app
is that this code is simply copied onto the device
and executed there.

That's not what's happening.
Instead, what is happening is that Dart in the end goes
through this code file top to bottom
and reads that code file from top to bottom.

It parses this code file, as it's also called.
So your code in your code files is parsed
by Dart in the end, when you are building your project
for running it on a target platform.

That's the first step.
Your code is parsed and analyzed by Dart,
because keep in mind,
Dart is the actual programming language.

Now, after your code has been parsed,

it must be translated to a language
that is understood by the target platforms
because as I just mentioned,
this code can't be executed like this on Android or iOS
or other target platforms.
Therefore, as a next step
after parsing the code,
the code is compiled by various Dart
and Flutter tools,
which simply means it's translated into some other code,
some native iOS or Android machine code.
So some code that is understood by iOS or Android
or whichever target platform you are targeting.
So the code you write with Dart
and Flutter is translated into some other code,
and that code is then in the end packaged up,
bundled together into a highly optimized code bundle.
And that code bundle is then executed
on the devices you are shipping your project to.
So on the mobile devices you are deploying your app to,
for example.
That is what happens to this Dart code.
So you work with Dart and Flutter,
but on the device, something else is executed.
And for that, the code which you write is parsed.
So is read and analyzed from top to bottom
before it's then compiled.
And in the end, it's then this compiled code
that is executed on those target devices.

4. How Programming Languages Work

So now that we know

that this main.dart file gets parsed from top to bottom,
and in the end, gets compiled to code
that is understood by the target platforms,
I wanna take a closer look
at what exactly we see in this file.
And this lecture is primarily meant for students
who have no or almost no prior programming experience.
If you did work with other programming languages,
you can skip this lecture here
because we're not going to dive into Dart
or Flutter specific features just yet.
Instead, I wanna generally explain what we see in this file.
So what do we see here?
We see a bunch of words, a bunch of lines of words,
not really sentences, just combinations of words.
And that in the end is what programming is all about.
It is about combining different words to create statements,
instructions that then tell the computer
or a mobile device in the end what to do,
what to show on the screen,
what to do when a button is clicked and so on.
Now, of course, here we just got some code
for a very basic app, which you saw before already.
It doesn't do anything useful,
but it does of course show something on the screen.
It, for example, shows a main title
and some text in the center of the screen,
and this code here is responsible for that.
Again, we're not going to dive
into what exactly which line of code does here.
We're not going to dive into the Dart or Flutter specifics,
but instead, I wanna take a more general look at that code.

And what you can see here is that we have that combination of words.

And what's important to understand about these combinations of words, which then create these instructions, is that you have two main categories of words you could say.

The first category are so-called keywords, words that are built into a language that do or trigger very specific things in that language.

For example, `import` would be a keyword built into Dart, which in the end connects this file to another file, but that's not too important.

But the other category of words that you have are identifiers, words that you as a developer define yourselves.

For example, `MyApp` here, we could have also named this `MeineApp`, which is the same in German.

I would have to change this in some other parts of this code thereafter, but we could use this instead of `MyApp`.

This does not have to be called `MyApp` here.

On the other hand, `import` does have to be called `import` because `import` is built into the language, `MyApp` is not.

Now throughout this course, you will of course, learn all these important keywords and you should definitely not learn them by heart.

You should not go to the official documentation and try to memorize all these keywords.

That's not how you learn a programming language.

Instead, programming is all about solving problems.

And in this course, you will learn how

to solve different problems and build different things,
and along the way you will learn and see
and automatically memorize which keywords are needed where
to do different things.

But it is important to be aware
of these two main categories.

And one of the main reasons
for using a code editor like Visual Studio Code is
that it helps you with detecting different kinds of words.

For example, like all major code editors,
Visual Studio Code has a built-in feature
called syntax highlighting,
which assigns different colors to different words.

For example, `extends` and `class` and `import` are blue
because they are keywords.

Now, it does not just use blue for that.

For example, `return`, which is pink is also a keyword
and the reason for that simply is
that different keywords can only be used
in different places,

and that's why we also have different colors here.

For the moment, it's just important to recognize
that different colors are being used
for different kinds of words.

Not one color per category,
but definitely different color
for different words and meanings.

For example, yellow
or this green light-bluish color here
would be used for identifiers.

Again, just as with the keywords themselves,
it's not important to learn

which color is used for which word.

That also wouldn't make any sense

because you can use different themes in Visual Studio Code

and that would then change the coloring of words.

But it's simply important to understand

that we have this syntax highlighting feature,

which exists simply to make code more readable

and easier to read for you as a developer.

If everything would be white, it would be harder

to quickly understand the meaning of code.

In addition, code editors also help you catch typos.

If I forget the A here, you see that the color changes,

and that immediately shows me

that something seems to be wrong.

And of course, I also get some red squiggly lines

to quickly show me that something is wrong

with this line of code here.

So that's never a reason for using code editors.

They help us catch certain kinds of errors

whilst we're writing the code,

instead of having to wait until we try to run it

and then see that it fails.

But if we go back to the code,

that's in the end, what code is about,

about combining different words, keywords, and identifiers

to create instructions that then do something.

Now, of course, you don't know yet

how these words can be combined.

You don't know yet which instructions can be written,

but that's of course, exactly what you're going to learn

throughout this course

as you learn about Flutter and Dart.

5. Starting From Scratch: Understanding Functions

So now it's really time
to get our hands dirty,
write some code and learn some Dart and Flutter.
For that, here, I'm in a new project,
but you can also stick to the existing project
that you have.
But you should make sure that in the lib folder,
in this main.dart file,
you delete all the content that's inside of it.
So that main.dart is an empty file
because we're going to write this app,
which we're building in this core section here,
from the ground up, step by step,
all from scratch,
so that we can really dive
into all these core Flutter and Dart features.
So how do we start in this empty main.dart file?
How do we give Flutter and Dart some instructions
to show something on the screen?
Well, it all starts with one key instruction
that you must have in every Flutter app you're building,
and that's the runApp instruction,
which is written like this runApp,
and then followed by parentheses and a semicolon.
This is a so-called function,
and functions are a core concept in Dart,
and actually in many programming languages.
Functions are simply instructions
that can be executed in your code.
And you will soon learn
how you can write your own custom instructions,

your own custom functions, therefore, as well.

Now, runApp is a function,

an instruction that is provided by Flutter.

So not written by you,

not built into Dart, the programming language,

but provided by the Flutter framework.

And this instruction,

as its name suggests,

is all about getting that app up and running,

or to be more precise,

its main job is actually to show some user interface

on the screen.

And that in the end is a vital task

for any app you're building.

Now, as you can see in this code editor,

I'm getting an error here though.

I do have this red squiggly line here,

which indicates an error,

a problem with the code.

And if you hover over such an error,

you get a more detailed description

of what's the problem here.

In this case, that a function body

must be provided.

Now, what's that?

Now, in Dart, it's actually not allowed

to run functions like this,

just like that in the main.dart file.

Instead, you must wrap instructions like this,

code like this, into other functions.

And for that, we must create a custom function,

a custom instruction.

And that's done like this.

You type void,

and I'll explain what this void thing means soon,

but you type void,

followed by the name of your custom command,

of your custom function,

which in this case should be main.

It must be main,

and I'll explain why it must be main in a second as well.

Followed by parentheses,

and then opening and closing curly braces.

That's how you define a custom function,

a custom instruction,

if you wanna call it like this,

in Dart.

This here, main is the so-called function name.

This here is the so-called return type,

and I'll get back to what that is soon.

And then this part here

between the opening and closing curly braces

is the so-called function body,

the code that should be executed

when this function is executed.

And that's another core programming in Dart concept,

functions, as mentioned,

are in the end instructions,

and you can create your own instructions, as you see here.

But those instructions, those functions,

are not executed by the mobile device,

by the computer on which the code runs,

unless you tell that device, that computer,

to run that instruction.

And you do tell a computer
to execute an instruction
by using that instruction's name
followed by opening and closing parentheses.
So in the end, what we're doing here in line six,
here we are executing a function.
We're calling a function, as it's also called.
Now, I did mention that `runApp`
is a function provided by Flutter.
So not written by you,
but provided by the Flutter framework.
And since Flutter is an open-source project,
you can actually inspect its code.
And attached you find a link to a file
that includes this `runApp` function definition,
which is exactly that function
that we're trying to call in our code,
so that we're trying to execute in our code.
And only if you use code like this,
you are actually executing a function.
So this `main` function, for example,
is at the moment only being defined,
but it's never executed
because we never have code
where we would call `main` like this.
If we would call it like this,
if we would add this piece of code,
we would execute this function.
That's why I also like to call functions code on demand.
You define them,
and then somewhere else in your code,
you can use them as often as you want by calling them,

as I just explained it,
by repeating that function name,
followed by opening and closing parentheses.
And of course, it's not just you who can define functions,
but also other developers.
Like the Flutter team did here
with the runApp function,
they defined for us as part of the Flutter framework.
And this runApp function must be executed,
but as we saw in this error description,
it must be executed inside of a function body.
So we should actually move it
into this main function body here,
like this.
In here, this function can now theoretically be executed,
but now we're getting a new error.
We're getting the error
that the function runApp isn't defined.
So in the end, Visual Studio Code
is telling us that it understands
that we want to execute a function called runApp,
but it doesn't find this function anywhere.
And where would it find it?
In our code file,
we only have the main function, nothing else.
Where should runApp come from?

6. Importing Features From Packages

So the code editor
doesn't find the runApp function.
It doesn't know where runApp comes from.
Now of course, I showed you that runApp

is defined in the Flutter framework.

It's coming from Flutter,

but when writing code, you have to be very explicit.

You have to tell your program

that runApp is coming from Flutter.

It's not guessing that this could be the case.

There are hundreds and thousands of libraries

and packages out there in the world

written by other developers, and therefore, of course,

your code and your code editor doesn't guess

from which of these many packages this runApp function

might be coming from.

Instead, you have to tell your code.

And for that, the first important step in a Dart program

is to go to the pubspec.yaml file

because in that file, you can manage the dependencies,

the third-party dependencies of your project.

You can reference other libraries

or frameworks from which you want to use code

in your project.

And in there, under dependencies,

you have to add all third-party packages,

which you want to use as dependencies in your code,

so from which you want to use features.

Now here, Flutter is actually already added as a dependency,

so we don't need to change anything there,

but it is important to understand

that this entry must exist

in order to use features from that package.

Of course, here we're still getting an error

despite having added this package in the pubspec.yaml file.

And the reason for that is that

even though we added this dependency,
our code still doesn't magically look into
those dependencies which we added,
and then check if there is some runApp function anywhere
in that third-party code.

Instead, we have to add a so-called import statement
at the top of the file.

And we do this by using a special keyword,
the import keyword, which is built into Dart.

The import keyword tells Dart
that we wanna kind of connect our code file here,
our main.dart code file here, with another code file,
or to be more precise that we want to use certain features,
certain functions, for example,
that are defined in some other code file,
in this code file here.

And for that, we have to add a so-called import path
after the import keyword
where we specify from which file we wanna import something.

You define such a path by adding opening and closing quotes,
double quotes or single quotes, that is up to you,
but single quotes are more common in Dart projects.

And then since we want to import some code
from a package here,
we start by writing package followed by a colon
and then followed by the package name.

Now you see I'm getting some auto suggestions here
by my code editor, which is a very useful feature
because it can speed up development
and it also reduces the danger of introducing typos
since we can use these suggestions
to auto complete our code, as I'll show you in a second.

But here at the moment,
I don't have any suggestion that I need.
Instead, I need some functionality from the Flutter package.
Hence, after the colon, we type Flutter
and you see that my suggestions are changing now
and then we want to get access to some code
that's defined in some specific file
in that Flutter package.
And hence we create a path to that file
by adding a forward slash.
And then for us, it's the material.dart file
from which we want to get code.
If I type MA here,
the suggestions are narrowed down even more.
And now as I mentioned,
it's this material.dart file I'm interested in,
so we can select this with the arrow keys
and hit tab to auto complete this path here.
And that's the part that can speed up development
and also avoid typos.
Now we must end this with a semicolon
because basically all your statements must end
with a semicolon.
Function definitions are an exception.
You don't add semicolons here after the opening
or closing curly braces,
but Visual Studio Code also tells us that
this semicolon here is unexpected,
so you can't really get this wrong.
It also tells you if you forget a semicolon.
Here, if I scroll down,
the error messages are always at the bottom.

I also get an error that a semicolon would've been expected.

And with that, we have a valid import statement.

And now the runApp error went away.

And you also might have noticed

that the color changed from white,

which is always a good indicator

that some identifier wasn't found to a light yellow color.

I still have an error here,

but that's now a different error, which we'll tackle next.

But now runApp is found

because material.dart in the end gives us access

to the runApp function to find by Flutter in their code.

And that of course was a lot of talking about this,

but understanding functions,

calling functions and function definitions

and how Flutter apps start is of course very important.

7. How Flutter Apps Start

So, in the last lecture,

we added this import path

and now this runApp function

is found and we added it to this main function.

But how exactly is this main function getting executed?

If you paid close attention,

you of course learned that functions can be defined,

but then they also must be called,

they must be executed in order to actually run the code

that's in their definition.

I mentioned before that if you just define a function

and you never call it,

the code inside of the function,

indeed also does not execute.

And at the moment, main is never executed
because we don't have this code here anywhere in the app.
And indeed, you don't have to add this code,
and you shouldn't add this code in line 7 to your app.
Indeed, that's not required,
because the main function is actually a special function
in Dart programs.

That's also why these strange instructions above it
were added automatically here in the Code Editor,
because the main function is the main entry point
of a Dart app.

To be precise, a main function in a main.dart file,
and it will be executed automatically by Dart
when your app becomes active on a device.
So, after your code has been parsed and compiled,
when it then executes on a device,
the compiled version of your main function
gets executed automatically on that device by Dart,
so you don't need to call main manually.

Instead, it is executed for you.

So, that's something that's taken care of,
and then inside of main, we of course, call runApp,
because as mentioned before,
that's another important step
when it comes to activating a Flutter app
or starting a Flutter app
and drawing some user interface onto the screen,
showing something on the screen.

Because runApp tells Flutter what to display on the screen,
so which UI elements should be displayed on the screen.

And that's now the last piece of code we must add
to this code here to have a valid Flutter app

that shows something on the screen.

I'm getting an error at the moment

that one positional argument is expected by runApp,

but that zero arguments are found.

And for that, of course,

it's important to understand what arguments are,

but also which kind of argument is expected here.

8. Understanding Widgets

So what are arguments?

Well, functions can be defined,

like the main function,

to take no input values.

But as you can see in the Flutter source code,

they can also be defined such

that they do expect some input values.

And that's the part between those parentheses.

Here you can specify a list of input values

that are needed by the function to work correctly.

Some functions don't need any input, like main.

Other functions do.

And these input values

are called parameters, or arguments.

Technically, they're called parameters

when we talk about them when defining a function

and arguments when we talk about them

when calling a function.

But I will simply use these terms

interchangeably in this course.

So runApp needs one argument, one input value,

to display something on the screen.

And this should make a lot of sense

because, like this at the moment,

how would runApp know

what to display on the screen?

Well, the input value it needs

is the content that should be shown on the screen.

So the user interface

it should show on the screen.

And for that, runApp needs a so-called widget,

or widget tree, as a value

because Flutter user interfaces

are built with widgets.

When building a Flutter app,

you don't use some visual drag and drop editor.

Instead, you build your user interface in code

by using so-called widgets.

You typically combine many widgets with each other.

You nest those widgets into each other,

as you will see over the next lectures,

to build a user interface.

And since you end up nesting widgets into each other,

you will actually end up with a so-called widget tree,

where you have some root widget at the top,

then maybe a child widget with another child widget,

which then may hold multiple child widgets

simply depending on which kind

of user interface you're building.

And of course, all these words here,

all these widgets on this slide,

don't tell you anything just yet.

But that will, of course, change

over the next lectures and sections.

So Flutter user interfaces are created

by combining and nesting widgets.

For example, by adding a text widget,

which is something Flutter gives you

to display text on the screen

inside of a button widget

to have a button with some text.

And Flutter provides many built-in widgets,

which you can use in your code,

like buttons from inputs, layout widgets,

but you can also build custom widgets.

And of course, we will also do that,

actually quite a lot,

throughout this course as well.

So runApp needs a widget

or a combination of widget,

a widget tree, as I mentioned.

So of course, the question is which widget,

and how do we add a widget in code?

9. Using a First Widget & Passing Values to Functions

Now when it comes to the question,

which widget should be used,

you can also search for Flutter widget catalog.

And you'll find an official documentation page which gives

you various widget categories, which you can browse

to learn about some of the most important widgets

offered by Flutter.

So the different building blocks you can use in your code,

for example, to display images, to display text,

and do many other things as well.

Now, please don't go to this page

and learn all these widgets by heart now.

That's not how you will learn how to build Flutter apps.
Instead, you will learn about all the important widgets
step-by-step throughout the course,
and you will see that in the end,
you will always end up using the same 10
or 20 core widgets with some occasional niche widgets
used for specific use cases.

But you will learn how to use them
and you will learn which widgets are important
simply by building apps
and simply by following along with this course.

Nonetheless, it's good to know that this page exists,
and here from this page, from the Flutter widget catalog,
we have to use at least two core widgets offered by Flutter
to have a basic user interface.

And the first widget which we want to use for that
which we therefore pass to `runApp`,
is the `MaterialApp` widget written like this,
capital M at the beginning,
and then capital A inside of the word.

This is one of the widgets provided by Flutter,
and we have access to it here because of this import.
So the same package that in the end gave us access
to `runApp` also gives us access
to all these core widgets like `MaterialApp`.

Now you then use a widget in code,
you create a widget, so to say, by adding opening
and closing parentheses after the widget name.

So `MaterialApp` is the widget name
and this now creates the widget.

Now this of course looks a bit like calling a function.
It looks like what we did with `runApp` before, right?

We also called that like this initially.

Now here we're calling MaterialApp.

And to some extent it is like a function,

but it's actually a so-called class

or constructor function of a class to be precise,

and I'll get back to classes

and constructor functions later.

For the moment, it's just important to realize

that we can create a widget like this in our code.

In this case, a built-in widget provided by Flutter.

And MaterialApp is a core widget,

which you will use in almost all Flutter apps

you're going to build because this is the main widget

which you use as a starting point,

so which you pass to runApp in the end.

Because this main widget does a lot of

behind the scenes setup work for your user interface.

So it ensures that the interface

can be shown correctly on the screen.

It does a lot of important

behind the scenes configuration stuff.

On its own, it doesn't show anything on the screen though.

Instead you need to pass another argument to MaterialApp

to show something on the screen.

And that's now really important to understand.

When calling MaterialApp like this,

we are creating such a app widget.

So we're in the end telling Flutter

to display this MaterialApp widget on the screen,

this part of the UI.

So the MaterialApp widget thing,

whatever it may be that is being created here,

is the value that's passed to runApp.

So which is passed as a value to this parameter thing,
this input value that is expected by runApp.

And by the way, I will get back to

what this widget thing here

in front of this app parameter means in the near future.

And that is something you'll see a lot in this course,
and you'll do a lot in programming in general.

When you define functions, so custom commands,
or you are using functions defined by other people,
like here where runApp is defined by the Flutter team,
many of those functions will take input values
to work correctly internally.

Here, this app input value is used inside of
that runApp function thereafter.

It's not important to understand what exactly happens
with it here, but it is used internally.

So being able to pass input values to functions
so that those functions can then do something
with those input values

is a crucial part of programming in general.

And here it's this MaterialApp widget thing
that's passed as an input value to runApp.

10. Positional & Named Arguments

Now as mentioned,

MaterialApp then also needs input value though.

Because like this, this MaterialApp thing
doesn't provide too much information

to Flutter and to runApp

regarding what should be displayed on the screen.

Instead, you must pass more information to MaterialApp

so that that information will then indirectly
also be picked up by Flutter.

And for that, between those parentheses here,
which belong to `MaterialApp`,
you must provide some input value to `MaterialApp`.

And that input value is now all about defining
what should be shown inside of that app UI
that will be displayed on the screen.

And for this, `MaterialApp`
also takes input value, a parameter,
though not a parameter that would be defined like this,
as is the case for `runApp`,
a so-called positional parameter
or positional argument.

Because here for `runApp`,
the first value you pass to `runApp`
is used as a value for the first parameter
in this parameter list.

And of course, `runApp` only takes one parameter,
but other functions could also take
multiple comma-separated parameters.

And then values would be passed
in a matched by position.

So the first value provided to a function
when calling that function
would be mapped to the first parameter,
the second value to the second parameter, and so on.

That's one way of accepting
and receiving parameters in dart.

But an alternative way,
which you will also learn in greater detail later,
is to define functions that take

so-called named parameters or named arguments.

When defining the function,

you wrap those parameters between curly braces.

And again, that is also something

you will learn about later

in the course in greater detail,

and which you will do a lot throughout the course.

But in the code where you define a function,

you would wrap those named parameters between curly braces.

And the difference compared

to positional parameters or arguments

is that now it's not important

in which order you pass values

to the function when you call it.

Instead, you identify those parameters by their name.

And that's exactly how `MaterialApp`

is implemented, internally, in the end.

There you have a long list of named parameters,

as you can tell by this opening curly brace

and that closing curly brace, thereafter,

that wrap all those parameters.

And those parameters are then simply identified by name.

Now regarding `super` and this,

these are all things we'll learn about

throughout the course.

But this `MaterialApp` thing,

which you can execute in the end,

is defined by using named arguments, internally.

And as a developer, it's up to you

whether you want to use positional

or named arguments when defining a function.

When using it, you, of course,

have to use what has been defined in there.

And here for MaterialApp,
it's this home named argument
that has to be set to display something on the screen.

And if you type here,
you, of course, also get suggestions.

Even without typing, if you press ctrl + space,
you'll get a list of suggestions.

And here you, in the end, see
all these named arguments that you could possibly set,
though many of them are, of course,
for niche settings or configurations.

The home named argument is important though
because that's the main argument
you must set for MaterialApp
to define which kind of widget,
which kind of UI element,
should be displayed inside of that app UI
that's passed to runApp.

And here I wanna display some text
by using the built-in text widget,
which is written like this,
and which is also executed like a function.

Just like MaterialApp, text is coming from Flutter
and provided by the Flutter team.

11. Combining Multiple Widgets

Now the text widget, if you hover over it,
also actually takes some named parameters.

You find a curly brace and a long list of parameters
between those curly braces here as well.

But there also is one positional parameter,

and that's possible as well.

You can mix and match those different parameter types
as needed when defining functions.

That is totally up to you, the developer.

And here, dispositional parameter is indeed
what we must set when using text.

Otherwise, we get an error
that one positional argument is expected,
but zero arguments are found.

And here, the one piece of data
that we must pass in is some human readable text,
which should be wrapped between single or double quotes,
and single quotes are more common.

And here, you could write Hello World, for example.

Now these single or double quotes are required
because without them Dart would think
that hello or world exclamation mark
should be keywords or identifiers.

Now hello and world are definitely not keywords
built into Dart.

So identifying these words as such fails
and they're also not valid identifiers,
because never in the Material Dart package
which we're importing, nor in your file, is there a hello
or world function or anything else
you could be referring to here, referring to run app,
material app and text work,
because those things are indeed defined here
in Material Dart.

Hello and world are not defined here and they shouldn't be,
because here we're not trying to execute some commands
or run some code.

Instead, this should be human readable text.

And whenever you have text that should not be interpreted as commands, you must wrap it with those quotes so that this is marked as text.

That is, for example, meant to be read by humans and not executed as code.

And with all those things we now have our first basic combination of widgets where we create the text widget and pass it to the material app widget, which then in turn is passed to run app.

And with that, if you now make sure that you have an emulator up and running, and you start your application on that emulator either by going to run, run without debugging, or by clicking this run button here, which is provided as a convenience by the code editor, by Visual Studio Code, you will see this Hello World text on this virtual device.

It looks ugly, it's definitely not a good looking app.

Also not a useful app, but it is a first step, and it's our first user interface built with flutter and widgets that's shown on the screen.

Now we can dive in deeper.

12. Understanding "const" Values

So we got our first screen output, our first basic UI, even though it's, of course, neither helpful nor pretty, but it's a start.

We achieved this by combining the MaterialApp and Text widgets

and passing that widget combination in the end to MaterialApp.

Now you might notice that there are some blue squiggly lines below that code.

Now, in Visual Studio code blue squiggly lines, as we have them here, mean that there isn't an error, but there is some improvement potential.

The code is not optimal, or it does something that should be changed.

And you can hover over the code that has these squiggly lines to learn more about the problem.

And if you scroll down in that info box, because the messages are always at the bottom, you see that here the recommendation is that we should use const with the constructor to improve performance, whatever const is.

Well, we can get rid of these blue squiggly lines by adding const in front of MaterialApp.

But what is this const thing?

Well, const is a keyword, as you can also tell by its color.

It has the same color as voided import.

So it's a keyword provided by Dart built into the Dart language.

And const is a keyword and a feature that exists to help Dart optimize the runtime performance of your application, of your application.

And how does it do that?

Well, if you mark something as const, let's say a Text widget that you're using in your app,

then this Text widget will be stored internally
in the memory of the device on which the app runs.
And that will always be the case
no matter if you use const or not.
Once your code executes the things that are created
by your code, the widgets, for example,
will be stored in the device memory.
But with const, when you then use that same widget,
that same widget with the same Text
a second time in the app,
that existing memory thing will be reused
instead of a second memory object being created.
So therefore, using const allows Dart
to reuse values that are indeed equal
and avoids the duplication of data in memory,
hence making your app more memory efficient
and improving overall performance.
Of course, behind the scenes things are a bit more complex,
but that is, in a nutshell, what's happening,
and that's why you should always add const
when prompted for it.
Now the good thing is you don't need to learn by heart
in front of which widgets you should add const, for example.
Instead, your code editor will simply tell you
by adding these blue squiggly lines here,
and then you should indeed add const.
Of course, for this basic app,
it doesn't really make a difference.
We're not reusing this combination of widgets anywhere,
so marking it as const
won't give us any big advantages here,
but it also doesn't hurt,

and it's simply a best practice to use `const` in front of all the values where you can use it. And as you saw, using it in the right places is very simple because your code editor simply tells you. It, for example, also tells you if you're using it in a place where you should remove it. For example, here, if I add it in front of `MaterialApp` and in front of `Text`, then I get a message here that I'm using `const` unnecessarily in this place because I'm already using it in front of `MaterialApp`, which then contains `Text`, so using it in front of `Text` in addition to that is simply redundant and therefore should be removed. But that's, in the end, what `const` does. It allows Dart to run your application in a more efficient way.

13. Building More Complex Widget Trees

Now, of course this is ugly as mentioned. So how could we make it more beautiful? Well, to make it more beautiful, we have to add more widgets. At the moment, we only have a very basic combination of widgets. One widget that will help us a lot with making this look better is another built-in widget, the scaffold widget. Now you can always of course read about widgets in the official documentation, and there you also learn how to use a widget. But you're watching this course so that you don't have to do that, though that of course

is always a great backup resource.

But the scaffold widget is a widget that in the end helps us with setting up a good looking screen in our app.

The material app widget is the root widget, which is important for setting up the overall app.

But many apps of course, consist of multiple screens, and those screens also must be set up.

And even if you have just one screen in the app as we have it here, this screen must be set up to have, for example, a nice background color to enforce some basic styling onto its child widgets and so on.

So therefore here, we wanna wrap text with the scaffold widget because scaffold must be inside of material app, but should be wrapped around the widgets that belong to a screen.

Now to wrap text with scaffold, we can of course type scaffold here and execute this and then grab text here, this text widget with its text content, and then pass it to scaffold, though scaffold now again wants a named argument.

In case you're wondering how I know which widget wants which kind of arguments, you can hover over a widget to learn about the accepted arguments here in this info window that opens up.

You can also learn about supported arguments here in the official documentation, but of course for the moment, I will simply tell you which arguments can be passed where,

and you will then automatically memorize these things
as you gain more experience with Flutter.

So for scaffold, it's actually a body
argument which we should set here.

And I'm also getting some auto completion here whilst typing
this, which of course helps a lot.

And the value for body then is the widget
that should be shown inside of this scaffold widget.

So with that, if I save this,
this should reload automatically.

If it doesn't, you can press this refresh
button here in this panel,
which opened up when you started the app,
and now it should update, and you see it looks much better.

We got the white background,
the "Hello World!" text looks nicer, even though it's still
of course hidden in the top left corner.

But we already gained a lot by adding this widget.
It's of course not enough though.

Instead it would be nice if "Hello World!" would be
kind of in the middle here
if it would be centered on the screen.

And that can be achieved with help of another widget,
which of course also can be found in the widget catalog
in the layout category.

Since I wanna center my texture,
so it's a layouting action I wanna perform.

And there it's the center widget that can help us
with, well, centering widgets.

For that, we need to wrap text with yet another widget.

And that is something you'll do a lot when
writing Flutter code.

You'll build these complex widget trees, as I called them before, where you have widgets inside of widgets, inside of widgets.

So here we wanna wrap text with yet another widget. And for that, we could manually add it as before, but you can also right-click on code here on this text widget and then choose this refactor action that's suggested by Visual Studio Code.

You can of course also use the shortcut that's suggested for that. And if you click that, you get some refactoring suggestions. And refactoring in programming simply is the process of changing your existing code.

And here you get some smart actions suggested. For example, you can choose wrap with widget to get some help with wrapping this with a widget, but you even get a wrap with center suggestion here, which you can pick by clicking on it or by selecting it and hitting Enter.

And this will automatically create the center widget and pass the text widget to center to be precise to the named child argument of the center widget. Now of course, things are getting quite unreadable here. This is becoming a super long line here, and this is still a simple widget tree.

And therefore, of course, you typically wanna structure this code a bit differently to make it easier to read and to maintain because even this basic widget combination is already getting too long here.

Now, to make this easier to read and more structured, Dart offers a nice trick

or actually recommended pattern

or approach to help with formatting widget trees.

For that you should add a comma

after every closing parentheses here except for the last one

where you need a semicolon.

And if you do that, you can run the format document command

in Visual Studio Code,

or use the shortcut for that to auto format your code,

which is something I do a lot when writing code in general.

And if you hit this shortcut, you see the code changes,

it's formatted differently,

and now it's way easier to understand this widget tree

because now you have some nice indentation

that shows you which widget is inside of which other widget.

And this widget tree, this code is now way easier to read

and understand.

And this is something you can do in many programming

languages, definitely in dart.

As you can see, you can split code across multiple lines,

and you then don't have semicolons at the end of those lines

because this in the end here is still one single instruction

just split across multiple lines.

Now, you can't split it at arbitrary points, for example,

not in the middle of a word, in the middle

of material app, that would fail.

But you can split it here

after these opening parentheses, for example, and also

after such a closing parentheses and such a comma here,

or even in front of the comma if you wanted to.

But this auto formatting shortcut will

basically do that for you.

And by simply adding these commas,

after all these closing parentheses,
you are giving the code editor in the end a signal where
to best split this code when using this auto formatting
shortcut because it will then simply do this
after every comma here.

And since you will build a lot
of widget trees when building Flutter apps,
and of course also way more complex widget trees
with way more widgets than we have it here.

It is absolutely a recommended pattern
to always add these commas after your closing parentheses,
and then use this auto formatting shortcut to make sure
that your code is structured
and easy to read and understand.

And if you save this now,
it again should update automatically,
or if it doesn't, as it's the case for me,
update it manually.

And now you see that "Hello World!" is centered in the
middle of this screen.

So by adding these two built-in widgets, scaffold,
and center, we again improved the app,
and we're making good progress towards our goal
of building our first custom Flutter app.

14. Understanding Value Types

So we built our first basic widget app
that leads to this user interface.

And before we'll dive deeper into that
and make it a bit more pretty
and of course work towards our final application goal.

For this section,

I wanna dive into another crucial Dart feature,
which is extremely important in which you must understand
in order to work with Flutter and Dart.

Because of course this section
is not about building the final app as quickly as possible,
but about learning all these important
Flutter and Dart fundamentals.

And one important fundamental Dart concept
that you must understand
is the concept of types.

Because Dart is a so-called type safe language,
which in the end means that
all values you are working with
are of certain types.

For example, this hello world text
is a so-called string.

If we had a number like 29 in our code,
it would be of type int.

MaterialApp is of type MaterialApp.

And actually there's not just one type per value,
but instead values are typically of multiple types.

For example, this hello world text
is not just of type string
but also of type object.

The number is not just of type int
but also of type num and of type object.

And MaterialApp, for example,
would be of type MaterialApp,
but also widget and also object.

So values can be of multiple types.

As you see, for example,
all values are at least of type object

and then they also have more specific types

and I'll get back to objects

and what's that all about a little bit later.

And those types are either built into Dart

or provided by a third party package

or provided by you when creating your own types,

which you typically do implicitly.

And in our application here,

we're also working with types

in a lot of different places.

But why does this type feature even exist?

Why does it matter that this text is

technically of type string?

Which you, by the way can also see if you hover over it,

because then the code editor shows you some information

about the thing you hovered over,

and that always also includes the type of that value.

For example, here string.

For a MaterialApp, it's a bit harder to see,

but here the type annotation would be

this thing in front of this

MaterialApp function argument information.

But why does this type feature exist?

It exists and it's enforced by Dart

to make sure that you're never accidentally using

the wrong type in the wrong place.

For example,

you might want to display a number on the screen

and therefore you could replace hello world

with that number.

But you would see that here in my code editor,

I'm now getting an error

and when I hover over this,
I learned that here in the end,
I can't provide a value of type int,
which is that number type,
which you just saw a couple of seconds ago on the slide
to a parameter that actually wants a string
so that once some text,
put in other terms,
the text widget internally
as it was defined by the Flutter team
only knows how to work with string input values,
with text input values.
It does not know how to work with numbers.
Of course, you could argue that
this playing a number on the screen
should be possible,
and it is if you convert this to a string
by wrapping it with quotes,
which in the end is what those quotes really do.
They create a so-called string text.
But of course numbers are a special case.
What if it would pass a MaterialApp widget here?
Now, it might make more sense that
text can't output this
because how should this MaterialApp widget,
which is all about creating the frame
for an application user interface,
how should that be output on the screen, right?
How should that be treated as text?
And to make sure that the text widget doesn't have to guess
what it should do with its input values
and how it should output those values as text,

the Flutter team built the text widgets simply such that it only accepts text, that it only accepts strings to be precise. And you can also see that if you hover over it to get more information about it. Here you see that first argument, the only positional argument, positional parameter wanted by the text widget, which is that parameter we're setting with that first value.

We're passing between those parentheses.

And here you see this annotation in front of data.

Data is simply the parameter name, but the annotation in front of it is the type that's expected for this parameter.

And if you would create a custom function which you want to use somewhere else in your application, you could also add such type annotations to make sure that you, for example, only accept numbers so-called integer, if you used the int value here.

So that for example, if you would pass some text here to this function, you would get an error because your function might not know what to do with text.

It wants numbers.

To add them as numbers, not to combine them as text.

That's why this type feature exists.

It's there to ensure that

you as a developer
can be clear about which kind of values
you want to accept
and which kind of values you don't want to accept.
For example, the runApp function,
which we're using in our code,
wants a widget as its first and only argument.
So a value of type widget.
And it turns out that MaterialApp,
which is in the end, the value we are passing to runApp
is such a widget.
And here for the text widget,
we should of course go back to a real text.
So that's the idea behind types
and in Dart, types are everywhere,
not just in the parameter lists of functions.
For example, this thing in front of main,
this void thing is also a type.
It's the so-called return value type of the main function.
But that is something we'll get back to later in the course
once we actually have functions
that do return something.
Now, as mentioned, you can build your own types
and you also get many types by third party packages,
but Dart also ships with many built in core types.
Of course, you might not understand all of them just yet.
You will see all of those and many more
in action throughout this course though.
For the moment, the most important takeaway
is that this types feature exists,
so that you can't work with the wrong types
in the wrong places.

15. Configuring Widgets & Understanding Objects

So, now that we know about types,
it's time to come back to our application code here
and continue working on this application.
And I would say one next step that makes a lot of sense
is to make this look a bit nicer here,
because to be honest, right now,
this application is really boring.
And for that,
I'd like to change the background color
from white to a nice gradient,
and also change how this text here looks like
so that it looks a bit nicer, a bit more fancy.
Now, let's start with the background color.
For this, we can go to this Scaffold widget.
And besides setting the body argument there to the widget
that should be displayed inside of the scaffold,
we can also set one of the many other arguments supported
by Scaffold because most widgets allow you
to pass values to different arguments.
You almost always have at least one argument,
like body or child, that wants another widget,
which then is included in the wrapping widget,
in the outer widget.
But you also often have other arguments
that don't want widgets,
but that instead allow you to configure
and or style this widget.
And for example, Scaffold,
if you add a new line in there and press Ctrl + Space,
supports a broad variety of arguments.
For example, the backgroundColor argument.

And as you might guess,
the `backgroundColor` argument allows you
to set the background color of this screen
that's created by `Scaffold`.
Now, background color wants a value of type `Color`.
Now that you learn about types,
you know that that thing in front of the parameter name
is the type definition.

And this wants a `Color`,
or actually it wants a `Color`?

Well, this question mark simply means
that we can either set a value of type `Color` or nothing,
so that this may be null as it's called.

We can set this to null,
which is a special kind of value and type supported by Dart,
which in this case, is pretty much the same
as not setting background color at all.

But if we do want to set it to some non-null value,
so to a value that has some visual effect,
it must be a value of type `Color`.

Now, such a color value can be created
with the built-in `Color` class or constructor function.

Now, keep in mind that I mentioned
before that `MaterialApp`, `Scaffold`, `Center`,
and `Text` are, of course, widgets,
but that technically what we're calling here,
like functions in our code,
are so-called constructor functions.

And I'll get back to the concept of constructor function
and related to that classes a little bit later.

Here, we can use the `Color` constructor function
to create a `Color` thing.

Though, if you're just getting started
with colors in Flutter projects,
it's typically easier to use the special Colors thing.
And then there, you don't add parentheses,
but instead, you use a dot
to access a list of predefined colors.
And here, you see a long list of predefined colors,
which you can use, like for example, deepPurple.
Like that.
And if you save that and force a reload,
you will see that color here
as a background for this screen.
And this already looks nicer
than the white background we had before,
at least in my opinion,
but it's not the final styling I wanna have here.
But what is this thing here now?
Well, if you hover over deepPurple,
you, for one, get a color picker,
which you could use to pick a color more conveniently,
which is pretty nice.
You will see that if you do that, though,
it changes from Colors.something to Color.,
and then some function that's being executed here,
to which some values are passed.
In the end, this here is a function
provided by this Color thing here,
which is provided by Flutter in the end,
which simply allows you to create a color
by mixing red, green, and blue colors and an alpha channel
that controls the transparency of the color.
So, that's simply a way of creating a color

by mixing three different colors.

Though, of course, the easiest way of using this and doing this is to use this color picker, which opens here in Visual Studio Code if you hover over it.

But, of course, that doesn't answer the question, what this here is in the end.

Well, if you scroll down from that color picker, you, in the end, see that this from ARGB function here, which it is, gives you a value of type Color, which makes sense because backgroundColor wanted a value of type Color, of course.

So, we got another type here, not a type built into Dart, but instead a type provided by Flutter, and what this in the end is, like all values as you learned before, is an object.

Widgets in Flutter are also just objects in the end.

They are objects,

and objects are the base value in Dart.

All other values are also objects in the end.

And objects are simply data structures in memory.

So, that's simply how Dart saves your values in memory on the device on which your application is running.

Now, objects are a core concept in Dart,

because as mentioned,

all your values are objects in the end.

And there are simpler objects like this Text,

this 'Hello World!' text,

but all the more complex objects like this Color object, or your widget objects like MaterialApp or Scaffold.

But in the end, all these are simply data structures in memory that are managed by Dart, and these data structures can also work together.

And for example, backgroundColor wants a Color object so that behind the scenes, the Scaffold widget object is able to reach out to that Color object and use that color information that's stored in that Color object to make the background color of this user interface purple.

So, it's all objects.

Widgets are also just objects, just special kinds of objects in the world of Flutter, if you wanna call it like this.

But all these objects are working together to bring the final user interface onto the screen.

And I'm really emphasizing this because when building apps with Flutter, you will always have many objects work together.

You, for example, have widgets that are nested inside of other widgets, but you also have configuration objects, like this Color object, which then, for example, could be used by your widgets.

And that's how we could add a background color.

Though it's, of course, not the final background color I wanna have here.

But using this helped us get started and become aware of the important concept of objects in Dart applications.

16. Working with "Configuration Objects" (Non-Widget Objects)

So knowing about this objects concept is crucial because all values are objects in the end as you learned.

But here, my goal is not to set this

just to some purple background color.

Instead, I want it to have a nice gradient.

Now this can't be achieved with `backgroundColor`

because that wants a single color.

And `Scaffold` also has no other argument that we could set that would allow us to add a gradient background.

Instead, what we can do here is we can add another new widget between `Scaffold` and `Center` so that it wraps `Center` and therefore implicitly also text

by right-clicking on `Center` and refactoring this.

And here we wanna wrap this with a container,

which is another widget provided by Flutter,

and `Container` is a widget that's very useful

for configuring styling and layout settings.

Now here, after adding `Container`,

I get some red squiggly lines, and if we hover over that,

we learn that we should remove `const`

because that seems to be responsible

for these red squiggly lines.

And indeed, if you remove `const` here in front

of `MaterialApp`, they go away and you can now re-add `const` in front of `Center`.

`Container` is simply a widget

provided by Flutter that doesn't support

setting it to `const`,

and therefore you also can't set some parent widget

that's higher up in this widget tree to `const`.

It is what it is, but your IDE clearly tells you

where you can and can't add `const`, so that's pretty good.

But of course we got more blue squiggly lines here now.

But the reason for that simply is that at the moment,

Dart and our code editor thinks

that we're unnecessarily using Container, and we do.

If we don't configure anything, it's a useless wrapper because if you save this and you reload, you will see that nothing changes about the app except for the background color.

Since I removed the background color on Scaffold, but nothing else changes.

But of course, our plan is not to do nothing here.

Instead, Container now also gives you many useful arguments that you can set.

For example, it provides a decoration argument, and this allows you to add various kinds of, well, decoration to the container and therefore also to the child elements inside of that container.

Now, Decoration wants a decoration argument, so a argument of type Decoration, and such a value can be created, for example, with the built-in BoxDecoration constructor function.

There actually also is a decoration option here in this suggestions menu, but this is actually not the right thing to use here.

Indeed, you can't use it if you try to use it because it's a so-called abstract class.

And we'll learn more about classes later.

For the moment, it's enough to know that you can't use that, but you can use this BoxDecoration constructor function, and this will then also give you such a decoration object.

Of course, it will give you a BoxDecoration object, but as you learned, the values can be of multiple types and BoxDecoration objects are also of type decoration.

Just as MaterialApp is not just of type MaterialApp,

but also of type widget.

This might all sound confusing,

but you're just learning it here.

You're just getting started with it.

You will get a lot of experience with that over time

as you're using all these features.

Now, this BoxDecoration constructor function,

which we're calling here

to build such a BoxDecoration object,

then also supports various arguments that can be set.

For example, the gradient argument,

which sounds just about right for what we're planning to do

because our goal is to add a gradient here.

So the gradient argument can be set here,

and that then wants a value of type gradient.

Now, there are different ways of creating gradients.

For example, a RadialGradient

or what I wanna do here, a LinearGradient.

This is another constructor function

that creates a LinearGradient,

which is also of type gradient

and which therefore is a value

we can set for this gradient argument.

And this LinearGradient constructor function

wants even more arguments.

And what we're doing here is quite typical

for Flutter development.

You often end up nesting widgets

and/or configuration objects into each other

so that they can work together

and give you the overall desired user interface.

Don't forget to add those commas

after all the closing parentheses to make sure
that your code is styled in a readable way
so that you never lose track
of how these things are related.
And here for a LinearGradient, there are again,
more things we can set.
Most importantly, this colors thing here,
this colors argument.
And this colors argument now wants a special kind of value.

17. Generics, Lists & Adding Gradient Colors

So, which kind of value
does this colors argument want here?
Well, if you hover over it, you see that it wants a list,
and then we have these angle brackets
and color between them.
What does this mean?
Well, for one, it means that it wants a list of values,
not just a single value, but multiple values.
That is what a list is in Dart,
a collection of multiple values.
And that makes sense,
because we wanna build a gradient here,
so a transition from one color to another color
or between multiple colors potentially.
And therefore, it makes sense that it's not enough
to provide a single color.
So, that's why we need to provide a list.
Now, these angled brackets which you see here
in the Type definition for this colors argument,
are some special darts and text,
which creates a so-called generic type.

This is also something we'll see
in greater detail throughout the course.
But in the end, a generic type is simply a type
that needs to work together with other types,
but is flexible regarding which exact types can be provided.
For example, List is a type built into Dart
that's quite flexible.

Depending on your use case,
you can have lists of numbers, of strings,
of other objects, whatever you need.
And when defining the type for a list,
it's simply the type between the angle brackets
that defines which types of values may be in that list.

And here, we want a list full of color objects.

That's what this type says.

So, therefore here, we now can pass such a list of colors,
and the list value is created in Dart
by using square brackets, opening and closing.

And between those square brackets,
you can now add all your values.

Now, for example, your strings,
but of course, here we don't want strings, we want colors,
so we can create them with the Color function
or with color from ARGB,
or with the Colors object,
which simply gives us some predefined colors to choose from.

So, for example again, deepPurple here.

Though here, I actually don't want that.

Instead, I'll use the color picker again
to select a nice, dark purple.

And you can, of course, choose any color of your choice.

And then in this list, we can add a comma

to add a second value to that list,
because in lists, values must comma-separated.
We can copy the first color as a second value in that list,
and then pick a slightly different color,
maybe more purple-ish, a little bit lighter,
whatever you want.
I'll pick this color,
and also add a comma after the closing square brackets.
That's another place where you should add commas
for proper code formatting.
And if I now auto-format this,
that's a bit more readable,
though it would also be nice
to get rid of these blue squiggly lines.
And here, if you hover over them
and you scroll down from that color picker,
you again see that it's about const again.
And we wanna add const at the highest possible place,
which in this case, is in front of BoxDecoration.
So, in the end, these are all, again,
objects that support being set to const
so that Dart can possibly cache, store these values,
and reuse them if we would use them
somewhere else in our code
to take advantage of this memory usage optimization,
and potentially improve our application performance.
And with all of that,
we're using a bunch of objects in conjunction,
and we learned about this special new list value type,
which is a generic type
that is able to work together with other value types.
Though here, it's really a list of colors that's needed.

And with all that, if you save that
and make sure that these changes get applied,
you see this gradient in the background.
And of course, you can pick colors with a greater contrast
to see this gradient a bit better.
But here, I have just that slight gradient,
which is exactly what I want.

18. How To Configure Widgets & Objects

So we added this gradient.
I am not totally happy with it yet, though.
I'm quite content with the colors,
but by default, that gradient goes from left to right.
Now I'd like to have it go from the top left
to the bottom right,
and this is something that can be changed
by adding more arguments to `LinearGradient`.
Besides setting the colors here,
you can also set the begin and end settings,
amongst other settings
with which you can of course also play around.
Now begin, once a value of type `AlignmentGeometry`,
which can conveniently be set
with help of the alignment object here.
Now you can set this to x and y coordinates,
or you don't use the parentheses here,
but instead the dot notation here
to access some predefined values again.
And here you have many predefined values like `center`,
`centerLeft`, and so on,
and you, for example, also have `topLeft`,
which is where I want my gradient to begin.

And I'll set the end argument to `Alignment.bottomRight`.

Also at comma here for proper code formatting.

With that,

if we save this and make sure this is applied,

the gradient now indeed goes from top left to bottom right.

And in case you're wondering how I always know

how to create the values that are wanted by end

and begin, for example,

so how I, for example, know about this alignment thing here

with those predefined values,

there are a couple of ways of finding out.

For one,

of course you learn it in this course

and you will learn about many comments,

super important widgets,

and configuration objects and options

throughout this course.

So that's a great start.

But in addition,

if you must provide a value and you have no value yet,

you can also always press `Ctrl+Space`

to get some suggestions.

And typically,

the first, or second, or a third suggestion

is already a suggestion that helps you

because this suggestion menu here is pretty smart.

And here for setting the alignment,

it even shows me these predefined alignments I could choose,

like for example, `bottomRight`.

So that's pretty good.

But of course you can also always dive

into the official documentation.

For example, for searching for Flutter,
and then the name of the widget you're using.
This will lead you to an article
in the official documentation.
And there you can also learn more about using this widget
or disconfiguration object or whatever you have here.
And you also often find some examples here
or a more detailed description of the constructor
and the AlignmentGeometry object.
And then here, for example,
you would get some hint about using the alignment thing,
which we did actually use.
So that's how you can learn more about this.
But again, you will see many common widgets
and configuration objects,
and options throughout this course.

19. Practice: Styling Text

So now that we get started
with some styling options,
I also wanna style this text,
and to be precise,
I wanna make sure that this text is white
and maybe has a slightly bigger font size.
Of course, we're soon going to get rid of the text anyways,
but as a practice, let's see how we could style it.
And even though it will be a challenging practice,
you also might wanna try this on your own.
See if you can find out how to change the text color
and font size of this text on your own.
Just as a hint, you won't need to wrap any widget around it.
Now here's a short pause for you to pause the video

and see if you can do this on your own.

If not, no problem.

After a few seconds, we'll implement it together.

So did you find a way of changing the color and font size?

If not, no problem.

This was challenging and we're just getting started here.

If you wanna style this text as mentioned,

you don't need to wrap any extra widget.

Instead, you just have to add some extra configuration

via some arguments,

so by adding a comma

and then adding more arguments

to this text widget constructor function here.

And that's, by the way, always a good way to start.

If you wanna style something

or change some configuration of some widget,

the first step should always be

to see if there are some arguments

that you can set that might help you.

For example, here, if I open this suggestion menu

after typing this comma and hitting Control + Space,

I see that there is this style argument I can set,

which sounds like exactly what we need here.

And then as a next step,

I get the suggestion here to add such a textile object,

as it turns out,

with help of the textile constructor function.

And this also sounds like a good idea, so I'll take this,

add a comma for formatting,

and then text style takes more arguments,

for example, a color argument,

which seems to set the text color,

since there also would be a separate background color option.

So I'll choose color here.

And color then wants a color object again.

And we, of course, already learned about that.

We can create a color manually, for example,

with this from ARGB function,

or we use a built-in one via the colors object here

and the dot notation.

And here it's white, which I want.

So that should set the text color to white.

And regarding the text size, the font size,

there conveniently is a font size argument I can set.

And this wants a value of type double,

and you learned a bit earlier

that this is in the end a number with a decimal place.

So here I can set this to 28, for example.

And you can even omit the decimal place,

though you can also set it, for example,

also to set this to 28.5.

But here I'll set it to just 28.

Add another comma for formatting.

And if I now save this and reload,

I should have a white text that is a little bit bigger,

and that indeed is the case.

And that's the practice solved.

That's how you can style text,

something we'll do again and again throughout the course

because text often needs to be styled in a certain way.

But that's, therefore, it for the moment.

Now it's time to dive into building custom widgets.

20. Onwards to Custom Widgets: Why Do You Need Them?

So by now, at this point,
our application is slowly but steadily taking shape
and it definitely looks way better than it did
at the beginning, but our code grew quite a bit.
Now, of course, that's still not too bad.
That's absolutely fine.
I'm also zoomed in quite a bit here to make
this all as readable as possible for you.
I can, of course, zoom out to make sure
that I really see all the code I wanna see.
But still, this widget tree with all
its extra configuration items is growing.
And therefore, what you typically wanna do
as you build Flutter apps is you wanna break
those big widget trees apart and build your own widgets,
which you can then use to split your code
into multiple smaller, potentially reusable building blocks
so that you can build your own widgets
by combining other widgets with each other.
And that's exactly what we're going to do now.
And, in general, when it comes to breaking up a widget tree
and extracting custom widgets there isn't
a single right or wrong way of doing this.
It's up to you which widgets you wanna put
into a custom widget but here, in this demo application,
I would say that it could make a lot of sense
to take this container with all its settings
and all its child widgets, so this entire selected part,
and put that into a separate widget
because that would then allow us to use this container
that has a background gradient applied to it

in other places of our application,
which could be a good idea because then we would have
that reusable prebuilt container.

Of course, in this application,
this is the only place where we are using it,
but in more complex, bigger applications
that might potentially have multiple screens,
it's quite likely that you wanna reuse certain widgets
across different screens or in different parts of the app.
And even if you're not reusing something,
breaking up this widget tree and putting this container
into a separate widget can still help us
with overall code readability and code maintainability.

So how do we put this into a separate widget, then?

Well, for that, we must create a so-called class
by using Dart's built in class keyword.

But what is a class now?

21. Understanding Classes

As you learned, widgets are objects.

And objects are simply these data structures in memory.

And of course, not just widgets are objects.

As you learned before,

all value types in the end are also objects.

So all value types are such data structures in memory,

and that's why these objects concept

is a key concept in Dart.

Now, classes are closely related to this objects concept.

For that, it's important to understand

that Dart is in the end, an object-oriented language,

because as mentioned, every value is an object.

And of course you have such primitive values

like text or numbers, which are created by, for example, wrapping text with single or double codes, or simply writing down a number in your code.

But you also have more complex values in your code, like for example, these widgets, but also these configuration objects, which you saw.

For example, the object that allowed us to configure our gradient or the individual color objects.

And those more complex values are created with help of classes, because classes in Dart, and in many other programming languages as well, are simply the blueprints for objects.

Now, why do we need such blueprints though?

Well, as mentioned, objects are these data structures in memory.

And here a good question would be, what's inside these data structures?

And the answer is data, so-called variables or properties.

For example, the username of a logged in user, or the color you chose for a gradient, but also logic functions.

Or as it's typically called in the context of objects, so-called methods.

So objects in the end are really all about organizing data and separating logic, so that logic that works with certain data is typically in the same object as that data, and not in some other object or some other place in your code.

For example, this color object here will contain the different color values for its red, green, and blue colors.

And if you add a dot thereafter,
which is the Dart syntax for accessing data or functions
that are grouped into an object, if you type such a dot,
you get a list of different variables, properties,
that you can access.

So the data I just mentioned
that would be stored in the object,
the red, green and blue color values,
and some other pieces of data as well.

But you for example, also get some functions down there
with blue, with green that, for example,
would allow you to create another color based on this color.

And which functions are provided by which object
simply depends on the definition of this object.

And that's now where these classes become important,
because with those classes, you define exactly that.

You define which kind of data will be stored in an object
when it is then created at runtime,
and you define which extra functions, so-called methods,
might be stored in such a object.

So color here, like container and scaffold
and box decoration, all these things are in the end,
classes, or when executed like this,
or when using this from ARGB function here,
you are in the end using
so-called class constructor functions,
which simply are functions that are used
to create objects based on those classes.

But all these things have classes as blueprints
as you can also see in the official documentation.

This is the linear gradient class after all,
as you can see here.

And you see more classes that are related on the left.
So these are all the blueprints that are built into Flutter
that you can use in your code to create objects
based on those blueprints.
So in your code, you in the end work with the classes
and their constructor functions,
and when your application then runs on a device,
and your code executes from top to bottom,
then the actual objects are created and stored in memory.
That's how you can think about that.
So all these built-in widgets,
like these configuration objects here, in the end,
are classes which you then turn into objects
by executing them like functions.
So by calling their constructor functions.
This process is also called class instantiation,
or object instantiation.
You are instantiating a class.
You are creating an object based on a class.
And that's why your custom widgets are also classes.
The built-in widgets are defined as classes.
You define your custom widgets also as classes.

22. Building Custom Widgets

So, now that we know
that it's all about objects
and that those objects are built based on classes,
the question is, which code do we have to write here
to create our own widget class?
And the answer to that is that you use the class keyword
to tell Dart that you are about to create a class,
and then any name of your choice, like GradientContainer.

The name is up to you, but it should follow certain rules.

For example, it should start with an uppercase character.

And if it consists of multiple words,

as this name here does,

these words must not be separated by white space or a dash,

but instead, you should separate them with an underscore,

or way better, and actually recommended it like this.

It's all one long word,

but the words inside of the word

start with another capital character.

So, that's then the name of your class,

and as mentioned, it is up to you.

It should just follow some general rules

regarding how the name is written

and it should also describe what the class is about,

so which kind of object will be created.

And here, my idea is to create a container widget

that will come with a gradient already applied to it,

so that name sounds quite fitting.

Now, you could create a class like this because in Dart,

you can also create classes that are not widgets,

and in that case, you would probably be done at that case.

You could add curly braces

and add your class logic between those curly braces,

just as you add function logic

between the curly braces of a custom function.

But since this should become a widget,

you actually have to add more in front of these curly braces

because classes can extend other classes.

They can inherit features and data

and logic from our classes.

And that's done by adding the extends keyword,

another keyword built into Dart,
and then the name of the class
from which you want to inherit.

And here, that's StatelessWidget, written like this,
which is a class that comes from the Flutter framework.
And that is available here because of this import,
which we added at the top at the beginning of this section.

So, StatelessWidget is the class
from which you want to inherit and inheriting from.
That widget does a couple of things.
Most importantly, it automatically, behind the scenes,
adds a lot of logic and data to this class here
without you writing anything between these curly braces.

It adds a lot of data and logic that is required by Flutter
in order to use this as a widget and add this to a UI.
But in addition to adding things behind the scenes,
extending from StatelessWidget actually also forces you
to add something to your class.

And that's why I'm getting an error here at the moment,
because at the moment, I'm not implementing everything
in my own class that I should implement.

Most importantly, I'm missing a method called build.
So therefore, to correctly extend this widget,
you should add a new method here called build
with parentheses and opening and closing curly braces,
so just as you would define a function

but now inside of a class,
and you can add functions to classes.

We then call these functions typically methods,
but it's, in the end, just a function added to a class,
so in the end, bound to that class.

And this build() method here must return a widget.

Therefore, you have to add widget in front of build() because as I mentioned earlier already, the thing in front of your function name is the return value type of your function.

Now thus far, we haven't created any functions yet that would return anything, but that's going to change soon.

Now, you also should add an annotation in front of your build() method, and that's @override.

This is technically not required, but you should indeed add it because this makes it clear that you are, in the end, overriding a method that is expected by StatelessWidget.

It's simply some extra metadata or annotation added to this method.

The more important adjustment that also must be made in addition to adding the type and @override, is that build() also must accept a parameter.

A parameter which you should call context or a ctx, though the exact name is up to you.

But this is required, because indeed, this build() method will be executed for you by Flutter.

Flutter will call this build() method once it finds your widget inside of some other widget or passed to the runApp function.

So, basically, whenever Flutter detects your widget in a widget tree, it will call this build() method.

And this build() method will then automatically receive by Flutter a value for this context parameter.

Now, for the moment, this context parameter doesn't matter. You will see it in action later in this course instead.

For the moment, you can simply think of context as a metadata object that contains some useful information about this widget in the overall widget tree.

For example, some information about where this widget is positioned in the overall widget tree.

Now, between those `build()` method's curly braces, you now have to add the body of this function.

And here, you have to do one important thing,

you must return a `Widget`

because `Widget` is the return type of `build`.

Now, up to this point,

in all the examples I just showed you from time to time, you always saw `void`.

That's, of course, also what we have in front of the main function.

And `void` is also the return value of a function,

but `void` is a special value built into Dart,

which simply means there is nothing to be returned,

so the function should not return any value.

And for `build()`, that's different.

`build()` should return a `Widget`.

Now, how do you return a value in Dart?

Well, inside of the function

or method that should return a value,

you use the `return` keyword,

and then thereafter, the value you want to return.

And when a function or method returns a value

with that `return` keyword,

that value will be available in the place

where the function was called,

and it can then be used there.

Now, in case of the `build()` method, as mentioned before,

it will be Flutter that calls this method for you behind the scenes when it renders the user interface. And Flutter will then take the widget returned by build() and make sure that it's reflected on the user interface. So, by using return here in this build() method, you ensure that Flutter gets access to the widget or to the combination of widgets you wanna store in your own custom widget here, because custom widgets, in the end, are all about storing combinations of other widgets and configurations. So, here in this build() method, you return the value that will be used by Flutter. For example, some text. Though, of course here, it's not some text that's wanted but a Widget. And therefore here, instead of returning text, we now go to our main widget tree and select the part of that tree that we wanna outsource. In my case, the Container widget with all its configuration and child widgets. And we cut that from here and return that in our own widget instead, in the build() method after the return keyword. Like this. Container in the end is of type Widget, and therefore, now here, Dart is happy because build() does return a widget. So, that's the first step. I got some blue squiggly lines here. For the moment, we'll ignore them. I'll get back to that soon.

But that's the first step.

As the next step, we can now use our own custom widget here in our main widget tree, which we pass to runApp.

And for that, we simply use our widget with its name, in my case, GradientContainer, and add parentheses thereafter.

And this calls the constructor function of this class, which in this case, is not a function we added explicitly. I'll come back to constructor functions in just a second.

But instead for the moment, we just call a constructor function that's automatically added to all classes.

So, even if you don't add your own constructor function explicitly, which you can do, but which we're not doing here, you get one automatically by Dart. And with that, if we save everything and force a reload, this app looks the way it did before because, of course, we haven't added any new UI elements.

But the fact that it looks like before and we don't have an error here shows us that outsourcing this code into a custom widget worked.

And this was a crucial step because you will work with custom widgets all the time when working with Dart and Flutter.

You will see it all the time in this course, and you will need this feature all the time in your own projects.

23. Working with Constructor Functions

So, we added our own custom widget.

Now, before we continue,
there are two things I wanna do with that.
For one, we got these blue squiggly lines,
and the question is, where's that coming from?
Well, if we hover over it,
we see that Constructors for public widgets
should have a named key parameter.
So, what's that about?
In the end, your own widget classes.
So, classes that extend StatelessWidget
or some other class which you'll see later,
should have a constructor function added to them.
I mentioned that typically you don't need to add one,
you get one automatically,
but if you wanna add some extra settings
or some extra configuration,
at the point of time this widget is created,
you have to add your own constructor function.
And adding a constructor function is quite straightforward.
You simply repeat the name of your class.
So, in my case, GradientContainer,
and add opening and closing parentheses thereafter.
Now, you could add opening and closing curly braces.
So, you add this like a method,
just without a return value,
but other than that, you added like a method.
And then here, you could do some initialization work.
This, by the way,
is a comment which can be added with two forward slashes
and which allows you to add some comment text
which will not be interpreted as code,
but which will be ignored,

but which can help other developers understand your code.

So, this is how you could add a constructor

and do some initialization work,

though often you won't even need to add these curly braces.

Instead, here, to get rid of these blue squiggly lines,

you just have to accept a special named parameter

as part of your GradientContainer.

And you do that by adding opening and closing curly braces.

This is how in any function,

you can accept named arguments.

Positional arguments are accepted

by simply adding a comma-separated list of parameters.

Named arguments or parameters are accepted

by using curly braces around that list.

And then here, for example,

A and B would be

the names of the arguments you wanna accept.

But here, it's a special argument named key,

which you should accept.

And this key argument, in the end,

needs to be forwarded to StatelessWidget.

So, this widget from which we are inheriting,

which we are extending, wants a key argument,

and that's why your class

should accept such a key argument as well

and forward it to StatelessWidget.

Now, to forward it, you have two main options.

You can add a colon after your constructor function name

and the parentheses,

and then thereafter,

you can do some variable initialization work,

which sounds strange,

also because we haven't learned a lot about variables yet.

But here, you can reach out to the parent class

from which you are inheriting

by using another special keyword built into Dart,

the `super` keyword, executed like a function.

Effectively, this is calling the constructor function

of the parent class then.

And here, you must set the named key argument

to your key here,

which is why I'm repeating key.

This key here is referring to my named argument called `Key`.

The first key in front of the colon

is referring to a named argument called `key`

in the super class in the `Stateless-Widget` class.

This would be one way of forwarding this key argument,

which is what you should do here.

But since you need to do this quite often,

basically for all your widget classes,

and this is a bit verbose if you write it like this,

Dart also gives you a shortcut.

You can simply write `super.key` here,

and this will accept a named argument called `key`

and automatically forward it to the super class,

so to `StatelessWidget` under that same name,

so also under the name `key`.

And that is something you should do here.

Now, I also got some more blue squiggly lines here,

where in the end,

I learned that I should turn this into a `const` constructor

by adding `const` in front of it.

And this is a place where we haven't used `const` before.

We only used it in front of values,

but in the end, when we used it in front of values,
we took advantage of constructor functions
that had this const annotation internally.

When defining your own classes
with your own constructor functions,
you often can add const in front of them
to tell Dart that this in the end,
is a class that can be optimized,
that can be stored in memory such that it is reused,
as I explained a couple of lectures ago
in this section already.

So, with const in front of your constructor function,
you basically unlock that const is then used
in front of your class when it's used as a value.

So, because of const added here, we can also add const here,
or since Scaffold and MaterialApp
actually also are defined to be optimizable with const,
you can even add const here in front of MaterialApp.

Using that optimization technique

I mentioned before in the course,
now here, just also with your class
unlocked to be optimizable.

And as always, the Code Editor will simply tell you
when you have an opportunity to unlock such optimizations,
in this case, by telling us that we should add const here,
which is why I'm doing that.

So, that was one thing I wanted to take a look at,
constructive functions, which is a feature
we'll later also use for our purposes,
but which we here needed to add
to make sure that this value,
which is wanted by the parent class,

is forwarded correctly,
and to unlock this potential optimization.

24. Splitting Code Across Files

Now besides constructive functions,
the other thing you should be aware of when working
with classes is that it's considered a good practice
to put them into a separate file.

So here we can select this entire class and cut it.

And then in the lib folder add a new file,
which I'll name, `gradient_container.dart`.

And the name is generally up to you,
but it should follow this naming pattern.

All lowercase, multiple words separated by underscores.
Dashes are technically allowed but will actually not work
and this would be quite unreadable, so underscore as it is.

And then in this file you can store your class.

And we do this, we split the code across multiple files
to keep every file readable and maintainable
so that all these files are relatively lean,
and don't contain endless amounts of code.

But you instead just have one main thing per file,
one main function or collection of functions
or one main class per file.

Now of course with that added to a separate file,
we got a bunch of errors in here,
because all these flutter specific types
and widgets are now unknown.

They are unknown because the import to the flutter package
is only here in the main dot dart file.

But this must be added to every file
where you are using features from that package,

which is why you should copy it

and add that same import here

in the gradient container.dart file.

And with that, all these errors go away.

We still have an error in the main.dart file

though because gradient container is unknown here.

This is not defined here,

because it is now defined in a different file.

And dart does not automatically scan and connect

all your files in your project.

Instead, you have to explicitly define any connections

between the files that you want to consider.

And you do this with the import keyword.

So here in main.dart, besides importing something

from the flutter material.dart file,

we also must add another import

where we also import from a package.

But this package is now our own project

and therefore, after package colon we use our project name,

in my case basics.

So here, all use basics.

This is basically whichever name you gave your project

when you created it with Flutter create.

So it might also be first app

if you're still working in that project.

Here for me it's basics.

And then here it's the gradient container.dart file

from which I want to import.

And you should of course, again, get some suggestions here.

Also regarding your project name.

You can always open those suggestions with controlled space.

And then here, if I import from gradient container dot dart

and add my semi gradient container is found again.
And hence, if I save everything and reload, it still works.
There is no error because we again
we have some valid code here.
Now following common best practices and conventions.

25. Practice: Create a Custom Widget

Now, when working with Flutter,
you will create custom widgets all the time,
so it's important that you're able to do this on your own.
Hence, here's your opportunity to practice this.

I want you to create a separate new widget
for this text part here.

This should be outsourced into a separate widget,
which you could, for example, name `StyledText`,
which you then use in this widget here,
in the `GradientContainer` again.

Create that custom widget for this styled Text
as explained in the last lectures
and after a short pause,
which I give you to pause this video
and try this on your own,
we'll outsource this into a custom widget together.

Did you succeed? Let's now do this together.

For that, I'll first of all create a new file
because you learned that you should put your own widgets
into separate files in that lib folder.

And I'll name it `styled_text`
because I will add a `StyledText` widget in there,
`styled_text.dart` it is, therefore.

And then, in there, we create a class
and I'll name it `StyledText`.

The name is up to you,
but it should follow these conventions
about which you learned.

And we of course extend StatelessWidget here,
though for this to work,
you also must add the import
to 'package:flutter/material.dart' as you learned.

Now with that set up,
we must add that build method
by adding @override and then Widget build()
accepting this context parameter
and returning a widget as you learned.

And the widget that should be returned here
should of course be this text here.

So, we can cut this from GradientContainer
and return this here inside of StyledText.

Now here, I got a bunch of blue squiggly lines
because indeed here we can now add const again,
as you learned if you hover over these blue squiggly lines,
since this can be cached.

This is, by the way,
independent from the fact
whether we add a const constructor to this widget or not.

This optimization in this widget tree
is detached from the fact
if we unlock this widget for being optimized itself.

But of course, here we should add a constructor
for the same reason as before,
to forward that key argument about which you learned.

Hence, I create a constructor
by repeating that name of the class.

Then here, opening and closing curly braces

to accept argument named key.

And with `super.key`,

we then forward this to `StatelessWidget`.

And here, we should now also add `const` since this is suggested to us here

to unlock various optimizations by Dart.

And with that, we have our `StyledText` widget defined.

We can now use it in `GradientContainer`.

There, we can type `StyledText` and use it like this.

But of course, to unlock it here,

we also must add an import

in the `gradient_container.dart` file.

Here, we must import from our own package,

in my case named `basics`,

in your case maybe named `first_app` or anything else, whatever you named your project.

And then, here I want to import

from the `styled_text.dart` file.

With that, with this import added,

this `StyledText` widget down here is found.

So, this here works

and if I save everything and force a reload,

my application still works.

And this was a nice practice for creating your own widgets.

26. Introducing Variables

So we learned about custom widgets.

A super important feature you're going to use all the time when working with flutter.

Now there's one thing you can do with custom widgets that makes them even more powerful.

But before we can explore that thing and

before we then thereafter can come back to this app
and add more features to that, we have
to dive into variables.
So what are variables?
Variables are data containers.
And this feature exists in basically all programming
languages because it turns out
that in all programming languages you are working a lot
with values, like also here
where in our styled text we have some text
or basically everywhere else, we have a bunch of widgets
and configuration objects, which are
of course also all values.

Now to understand variables, let me go back
to the gradient container here
and here we could say that maybe these alignments, which
of course also are values,
should not be stored here in this part of the code,
which is hidden quite a bit inside of the widget tree.

But maybe we wanna define them up here at the beginning
of the file so that if we ever want
to change these alignments,
we can do it right here at the top
of the file without having to dive into our widgetry.
This is of course just a convenience thing in the end,
but the more complex your code and project gets,
the more little tweaks like this could help.

To do this we could cut the value from down there
and instead store it in such a variable
because variables as you learned are data containers.
In dart they are created with help of the var keyword,
then the equal sign operator,

the so-called assignment operator,
and then the value that should be stored,
for example, a string.

So here, if I wanna store this alignment value,
I could create a variable here outside of my class.

We can also create it inside of the class,
but for the moment let's do it outside,
then give it any name of your choice.

Like for example, `startAlignment`.

The name is up to you, but just as
before with other names,
it should not be separated into multiple words.

Instead, words inside of words should start
with an uppercase character, but unlike class names,
the overall variable name should start
with a lowercase character.

So it should follow this naming pattern,
and the name should describe which kind
of value will be stored in
that data container in that variable.

And then we assign the value with an equal sign,
in this case, `alignment.topLeft`, for example.

So that's how we could define a variable.

We could then add a second variable,
and that is this `endAlignment` variable.

So therefore I'll name this one `endAlignment`
and store `Alignment.bottomRight` and there.

And with those two variables added,
we could use them down there to assign `startAlignment`
as a value to begin and here `endAlignment`
as a value to end.

Though here you will notice

that I'm now getting an error that a value
of type null can't be assigned to a parameter
of type AlignmentGeometry.

This is a bit cryptic to understand though.

What does it mean?

Well, variables when created with the var keyword,
which we're doing, here can actually be reassigned.

Here I'm assigning an initial value,
but I could then change that value.

For example, here in the build method.

Here, I could set startAlignment to Alignment.center.

That's a core mechanism of variables created with var,
you can reassign them.

Now, that's in the end where this error is coming from.

This null part can be confusing,
but it actually also tells us that the const keyword seems
to be the problem that we might wanna remove
this const keyword,
and here it's referring to this const keyword in front
of BoxDecoration.

Since if we add cons here, it's also implicitly added
to LinearGradient,
and that is the object that takes begin and end.

Since the values of these variables could be changed,
it's not guaranteed to be stable, it's not guaranteed
to be constant,
and that's exactly the kind of problem we're facing here
if we do have const in front of LinearGradient
or BoxDecoration, we can't tell Dart that it can cache
and reuse that value

because actually these variables are not locked in.

They could change potentially, and

therefore reusing this value would be unsafe
because Dart might accidentally reuse an outdated value,
which is why you can't use const here then.
You could add it in front
of this list still since these colors are indeed locked in
place here and hard coded here.
But that's the idea behind variables.
We simply have data containers that store values
and we can define them once
and then use them multiple times wherever we want
later in our code
as long as that code has access to the variable,
which here in this case is the case.

27. Variables & Types - Combining Two Key Concept

Now variables actually also have types.
Here, I'm creating two variables,
and in this case here,
Dart is able to infer their types.
If you hover over your variable names,
you see the type here,
and Dart knows which type will be stored in this variable
because it simply takes a look
at the value you are initially assigning to the variable
and uses that values type,
in this case here, Alignment,
as a type for the variable.
If you would not assign a value initially here,
and you instead would only set one
at a later point of time,
like this, for example,
the inferred type would be dynamic,

which is a special type in Dart
that, in the end,
just means that Dart knows nothing
about the type of this variable,
and it will indeed accept all value types.
Typically, you wanna avoid this
because that can lead to bugs in your application
because without type information,
Dart can't ensure
that you're not accidentally using the wrong type
in the wrong place.
And therefore, in such cases
where you don't assign a value initially,
you should replace the var keyword
with the name of the type
that will be stored in that variable.
Here, for example, Alignment,
with that, Dart would have more type information.
Though, of course, here, we would still get an error
because we tell Dart
that we wanna store a value of type Alignment
in this variable,
but initially, we store no value in here.
We would have to add a question mark
after Alignment to make it clear
that startAlignment is either of type Alignment or null
so that it's basically optional whether it's set or not.
So a null value would be allowed here.
Now this would lead to more errors down here
because begin actually doesn't want a null value,
and therefore, I'll get rid of that
and just initialize it as I did it before,

all in one step.

But it is important to know about this alternative syntax

where you do explicitly set a type

and not rely on type inference

if you're not initializing the variable with a value.

If you do that, as we're doing it here,

using the `var` keyword is recommended

because here you should indeed rely

on Dart's type inference.

28. "final" & "const" - Special Kinds Of "Variables"

Now, there is more

you should know about variables.

For example, you sometimes have variables that never change.

Of course, technically, it's called variables

because the value is variable,

but in the end, it's all about having data containers,

and those data containers could also be set once

and never changed thereafter.

For example, here, I never assign new values

to `start` and `endAlignment`.

And if you know that this is the case,

it's recommended that you use `final` instead of `var`.

`Final` is a never built-in keyword,

which simply means that this data container

will never receive a new value.

And whilst you could, of course, stick to `var`,

it's always a good practice to be as restrictive as possible

so that you never accidentally reassign a value here.

Of course, you are the only one

working on this project here probably,

but in bigger teams

with multiple developers working on the code,
you wanna be as restrictive as possible
to make sure that some colleague of yours
is not accidentally overriding
one of your variable values
somewhere else in the overall project code.
That's why the final keyword exists.
Now, here, I'm of course, getting some blue squiggly lines
that I should use const instead of final in the end.
And indeed, that's another usage of const
which we haven't seen up to this point.
You can create so-called constant variables
or constant data containers, you could say.
It's almost the same as final,
because const also makes sure
that you can't reassign those variables.
But unlike final,
it provides some extra information to Dart.
When using const here in front of a variable name,
so on the left side of the equals sign in the end,
you're telling Dart that the value that's stored in here
is a so-called compile-time constant,
which means it's simply locked in at the point of time
this code is compiled.
Now, that's not necessarily always the case.
You could, for example, also have some code
where you're maybe calling a function, getAlignment,
which doesn't exist in this project,
but which could exist,
and which then behind the scenes,
could dynamically calculate the desired alignment.
You could definitely have something like this

in your application,
and if that's the case, that of course,
means that this function must be executed when the app runs
to get the actual alignment value.
In that case, it might still be final.
You don't reassign the variable thereafter,
but it's not compile-time constant
because the value that might be returned by `get alignment`
is not known at the point of time
when this code is compiled yet.
Instead, it's only known once the code executed.
And here that's different for `Alignment.topLeft`.
This indeed is locked in
at the point of time this code is compiled.
`topLeft` is clearly defined
to the top left corner of the device,
so to certain X and Y coordinate values.
And therefore, you can use `const` to tell Dart
that this is not dynamically derived at runtime,
but can be locked in and set at compilation time.
And this is another potential performance improvement,
because if some value can be locked in
at the point of time the code is compiled,
this code doesn't have to execute when the app executes.
Instead, this locked in
and internally saved value can be used by Dart
when this code is about to execute.
So, it's some internal performance optimization,
and therefore here, when defining variables,
you should also use `const` instead of `final`
whenever it makes sense.
So, whenever you have such a compile-time constant value.

And just as before, the code Editor will simply tell you when you have such an opportunity.

By the way, with that set to const, you can also re-add const down there in front of BoxDecoration, because now that this is const instead of var, you guarantee that it will never be reassigned, and therefore, you can use const here, because this code is now no longer potentially variable because it can't be set in any other place.

It can only be set once, and therefore, here, we can unlock this potential optimization of the BoxDecoration object again.

And in case this is all not 100% clear just yet, you will see variables and final values and constants over and over again throughout this course, but it is important to be aware of these concepts right now already, because we'll use more variables in the next lectures.

29. Instance Variables (Properties) & Configurable Widgets

So now that we learned about variables, let's use this variables feature to make some of our code more reusable, like for example, the styled text widget.

We created a custom widget here so that we have this reusable style text.

But at the moment, this is only partially reusable, if we're honest, because 'Hello World!' is hard-coded in this code snippet here.

So whenever we use style text anywhere in our application,

it'll always output 'Hello World!'

Now the idea behind this custom widget,
it is probably that we want to be able
to reuse this configuration for this text,
so the color and the font size,
but the text itself should be dynamic.
Put in other words, it would be better
if in gradient container where we use style text,
we could pass the text to style text
so that this can still be set in the place
where we want to use our custom widget
instead of having it locked into our custom widget.

And that's of course, a super common use case
because most of your widgets should be reusable,
and should be flexible,
and should be able to receive data from outside,
like here where we pass data into styled text
from inside the gradient container widget.

So how can we make our own custom widget reusable then?

Well with help of the constructor function.

Because I mentioned before that here, when you use a class,
your own one or one provided by Flutter, doesn't matter,
with parentheses like this, which is how you have to use it
in order to turn it into an object,
you're calling the constructor function of that class.

You get one automatically if your class doesn't have one.

Otherwise your own constructor function is called.

Well, and since this is just a constructor function
being called here,

if we want to be able to pass input values
to that constructive function, we have to accept them here.

We are actually already accepting this named key argument,

because we had to, to forward it to StatelessWidget,
but we can accept more than just that key,
either more named arguments by adding commas here
and adding all the named arguments that we want to accept
between these curly braces,
or in addition to these named arguments,
also positional arguments before those curly braces.

So for example here, we could accept the text.

We then should also add a type annotation here
to make it clear which type of value this should be.

Otherwise it's dynamic

and we typically wanna avoid that, as you learned.

And here it would be some string, which we want some text.

And with that,

we now are able to pass some text to style text.

This works.

However, with that, we are not using this text in here.

We are accepting the text here,

we're passing some text to style text,

but this text argument,

which we are accepting in our constructor function,

is now not being used here.

Instead,

we still have the hard code 'Hello World!' down there.

So how can we use this argument?

Well not by using its name here.

If we try to do that and we use the name text down here,
we get an error.

We get an error because this argument,

which we accept in the constructor function,

is not magically available

in some other method of the class.

These two things are detached from each other
even though they're in the same class.

This is just how that works.

Instead, what we have to do

is we have to add a so-called class variable,
a so-called property to this class.

So a variable, just like the variables we added before,
but now not outside of a class,
but instead inside of the class to which they should belong.

So here for example, a variable, called output text,
or just text, it can be the same name as this argument.

That's no problem.

So we can add this, and if we don't set it to a value,
as you learned, you should instead define
the value type here instead of using var.

So here I'm just writing string text,
and with that, we added a variable to this class,
and this variable is now available in all the methods.

Of course, I'm still getting an error down here,
but here the error

is now actually a different one than before.

If I remove this text variable

and we take a look at this error,
we see that now it doesn't even find anything called text,
which makes sense, because as mentioned,
there is nothing named text.

This argument here doesn't count
because the constructor function
and this method is not connected.

If I add this variable here, I get a different error.

It's no longer saying that text is not known.

Instead it's, well complaining about some other things here.

The reason for that is that now we do have this variable,
but in the end, this variable is still not set
to the value we receive as a argument here.

To do this, we got two main ways.

We can again add this colon here after our argument list,
and then here we could refer to this text variable
by its name and set it equal

to this text argument we're getting here.

So text equals text might look a bit strange,
but the first text here refers to the text variable.

The second text here refers to the text argument.

So if I use different names here,
like output text here and here, this becomes clearer,
and I would also have to use it down here.

So now we would be setting our text
to this argument which we receive.

However, as before with super,
this is unnecessarily verbose,
and therefore Dart gives us a shortcut
because this is a very common pattern
that you receive a value on an argument
and you wanna store that in a variable
that belongs to the class.

You can replace this string part here with this dot
so that this says this.text.

Now you just must make sure
that your variable is also called text,
and also down here of course.

This is simply a shortcut
which looks for a variable called text,
so this variable here, and sets it equal
to the argument value that is received here.

So in this case, the first argument that's received
by this constructive function.

The `this` keyword is another keyword provided by Dart,
and `this` is a keyword that's used inside of classes
to refer to the class itself, or to refer to the object
that will be built based on the class,
and we can use it here to basically tell Dart
that we want to access the text variable
that is defined in this class, so to say.

Now, typically in Dart, you don't need to use `this`.

If you just use the name of a class variable,
as we're doing it down here in `Build`,
Dart automatically looks for a class variable
with that name, but here for this shortcut
in the constructor function, it is required,
because if this would not be added here,
you would just be telling Dart
that you want to accept some positional argument,
which you named `text` here in this constructor function,
but it wouldn't automatically look for
an equally named variable in the class.

And by adding this dot in front of it, you instead tell Dart
that you don't just want to accept this positional argument,
but that instead it should also automatically be mapped
to a class variable with the same name.

Now I'm just getting an error
because we have to remove `const`
because now that we receive some `text` here,
which is stored in a variable called `text`,
this can no longer be marked
as a potentially constant object
because the `text` might be changed thereafter,

because this is a variable,
and we also have to get rid of const down there
for that same reason.

We could still turn this configuration object here
into a constant though,
because this is still all hard coded,
but that is how we could accept such a argument,
store it, and then use it.

However, here we also have a great example for using final
instead of setting this to be a variable,
because this variable here only exists
to store that value which we receive on this argument.

We never planned to reassigning it
inside of this class then.

And therefore you can actually also add final
in front of the type here, not instead of the type,
because we still want to be clear
about which type of value will be stored here,
but you can add it in front of the type
to make it clear that this will only be set once
by this argument that is received,
and will not be changed thereafter.

Well, and with this done, you can re-add const here
because now this class can be optimized again,
because since this will never change internally,
this is guaranteed to always be the same object
once it has been initialized.

Hence, it can be cached and reused by Dart.

But that's how you can accept arguments
and store them in instance variables,
so in variables that belong to these classes
and the objects that are based on these classes,

and that allows you to reuse style text
with potentially different text values
so that the text itself
is no longer hard-coded inside of this class.
And again, that were a lot of new concepts,
a lot of talking about that,
but it is never a crucial feature
because being able to accept and store
and use values in your own classes,
no matter if they are widgets or not,
allows you to build highly reusable classes and widgets.

30. Practice: Reusable Widgets & Constructor Functions

So now that we adjusted this class
to be more reusable
by dynamically accepting input values,
you can do the same for the GradientContainer
as a little exercise here.
At the moment here,
we got some hard-coded colors.
And whilst this of course works here
and is absolutely what we need for this demo app,
if this would be a bigger project,
a bigger app, potentially with multiple screens,
we might want to be able to reuse this GradientContainer
in different parts of the app with different colors.
And at the moment, that would not be possible
because the colors are locked in here.
Therefore, as an exercise,
you should use the new knowledge
you gained in the previous lecture
and make GradientContainer more flexible

by basically ensuring that when we use GradientContainer,
we can pass our colors here as arguments,
in whichever way you want, to GradientContainer
so that inside of GradientContainer,
those color inputs are accepted, stored,
and then used down there.

There will be different ways of solving this,
and I'll give you a short break here
to pause the video
and try this on your own,
before then thereafter,
we'll solve this together.

So did you succeed?

Let's now solve this together.

As mentioned, there will be different ways of solving this.

For example, you could accept a colors argument here,
a positional one or a named one.

Doesn't matter, it's up to you.

And here I'll go for a named one
to show you both approaches.

And this colors argument, of course,
should be of the same type as the colors down there.

So it should be a list of colors.

We can simply copy this type here
to set this as a type.

Though, as you learned,
we can even switch to different syntax
since we'll need to store this in a variable anyways,
and add our variable here,
or better, our final variable,
our final data container of type List Color,
which could be named colors,

and assign the value

that's received on this named argument here

to this parameter

by using `this.colors` here.

Now, if you tried to solve this with a named argument,

you likely encountered this error here,

which in the end tells you that the problem now

is that this is optional

where it actually shouldn't be.

The reason for that is that named arguments

by default are optional with Dart.

You can solve this

by adding `required` in front of this.

Now, most likely, you simply went for a positional argument

because that's easier to use

and that's what we used before,

and that's of course totally fine.

I just also want to show you this alternative

that if you wanna use a named argument,

you can do this as well,

but you have to add `required` in front of this.

And here I'm keeping this approach

so that you now know also about this alternative

and how you can use named arguments

that are actually not optional

because by default, as mentioned,

named arguments are optional.

So now with that, we got our `colors` input,

we're storing it here in this variable,

and we set this to exactly the type we need.

And then, of course, we could use the `down` there

as a value, like this.

Now, I have to get rid of const here
because colors here actually is a list.
Even though it's final, it's still a list.
And lists by default in Dart can be edited.
They can be mutated even if they're final,
which is a bit more advanced
and not too important right now.
But for that reason, you can't have const down there.
And you can see this if you hover over colors down there,
you in the end, get that hint
that you should remove the const keyword.
So that's what we have to do here, then.
That's one way of solving this.
Now, I'll copy this entire widget
and comment it out
so that we can still see the code,
but it's not active anymore.
And I'll paste it in again above the commented-out widget
to show you an alternative approach.
Because of course, instead of accepting
such a list of colors,
you can also accept two individual colors.
You could, for example, accept two positional arguments
or two named arguments,
totally up to you.
color1 and color2,
that could be the names.
And both should be of type Color, of course.
Then we would need our variables,
or again, better our final values here,
which should be of type Color,
color1 and color2.

And now we can use these shortcuts

with the `this` notation here,

just as you learned before.

Again, this would accept some input values

and store them in some variables.

Now the difference is that we don't accept a list

of any kind of values

and any amount of colors,

but instead exactly two colors,

which are both required

because positional arguments by default are always required.

And now we can use these colors,

`color1` and `color2`,

down there in this list.

And with that, in `main.dart`,

you can then pass your colors

to `GradientContainer`.

If you took the first solution

with that list of colors,

you would pass your list of colors here.

If you took the second solution instead,

you would pass your two colors here.

So again, here, I'll just quickly use `deepPurple`

as a starting point,

and then use this color picker

to pick some dark blue, purple-ish color.

And then copy that, add a comma,

and set my second argument here,

which is also color,

which is a lighter purple color,

like this may be.

And with that, I'm applying these new colors

to this application here
now with my reusable GradientContainer component.
And again, this component here is reusable
because I'm accepting these color values
dynamically as arguments,
I'm storing them,
and then I'm using them in my build method.
And this is a pattern, as mentioned,
which you'll see a lot throughout this course,
that you're building reusable widget classes.

31. Displaying Images & Using Multiple Constructor Functions

Over the last lectures,
we had a look at many new features,
many core concepts you use a lot when working with Flutter,
but now it's time to continue working on the actual app
and what we see there.
And of course there the goal is
to have a button which we can press to roll a dice
and therefore it makes sense to, as a next step,
show a dice on the screen, show some dice images.
For that attached to this video,
you find a zip file which you can download and extract,
and in there you find a couple of image files.
Now you should add those files to your project in order
to then use them in your code.
For that, in your project folder, add a new folder,
which can be named assets.
This name is up to you,
but assets makes sense since we're about
to store some images, some assets
that we'll use in this application.

And in this assets folder,
you could even create a sub folder called images
in case you had other kind of assets
in this project as well,
which we don't have here,
but still it doesn't hurt to have this sub folder structure.
Now you should drag and drop all those image files,
which were attached into this images folder
in your assets folder.

And in order to use them in your code, in your project,
then you also have to go to the pubspec.yaml file.

And in that file, scroll down
until you find this commented out assets part here.

Now you can comment this in
by simply removing these hash symbols here
and you should make sure that you have one level
of indentation here by hitting the tab key.

And then you add a list of all the assets, all the images,
for example, that should be available in your application.

Just adding these files in your project folder
is not enough.

If you want to be able to display these images in your app,
you have to register them here.

And for that you have to specify a relative path,
the relative path to the images that should be included.

So in my case assets, since I have this assets folder,
/images, since I have that images sub folder,
and then the name of the images, dice-1.png,
but then also dice-2, dice-3, dice-4, dice-5,
and dice-6.png.

All these image paths should be added here
to assets in your pubspec.yaml file.

Now with that added, you can use those images in your code.

And here we could do this in gradient container instead of this text,

because this text here is not needed right now.

Instead here I want to output one of these images and later we will add logic to change this image dynamically.

Now to output an image, we can use the built-in image widget, which is provided by Flutter.

You can create it like this to then set an image, but for that you need a so-called image provider, which is another object, which in the end resolves to a certain image.

This could be done, but if you have a locally stored image, so such a asset which we just registered in a pubspec.yml file, there is a quicker way of referencing that.

You can use Image. and then the asset constructor function.

Now this is using a feature built into Dart where you can define multiple constructor functions in a given class.

We could do this in our own classes as well.

For example, here in GradientContainer, we have this default constructor function, but we could also add a second constructor function by again using our class name and then dot, and then the actual constructor function we want to use.

For example, purple to then, for example, set color one to some hard-coded deep purple value like this, and then separated by a comma.

You could and should also set color2

to some predefined value like `Colors.indigo`.

Then you could add cons here

and also accept that named key parameter.

And now you would have an alternative constructor function

that can be used whenever a gradient container is used so

that instead of using it like this

and passing those specific colors to it,

you could use the purple constructor function,

which here could act as a convenience constructor function

with some predefined colors.

And it's totally up to you what you would want

to do in such additional constructor functions.

Here, I'll go back to the previous usage,

but it is important to understand

that you can simply add multiple constructor functions

to the same class.

And that's in the end what we're using here

with this asset function.

`Image` has a default constructor function used like this,

and then multiple alternative constructor functions,

which simply make adding an image a bit easier.

For example, here,

when using the asset constructor function,

you pass a string to it, the name of the image,

or to be precise, the path to the image,

which in this case would be `assets/image`,

and then for example, `dice-2.png`.

Now here, I then get an error

because actually I should remove the `const` keyword

because this image widget is in the end a dynamic widget

which isn't guaranteed to be `const`.

So we must remove this `const` keyword to avoid errors.

But with that, we have this image here,
and since we now got rid of that styled text,
we can also get rid of this import.
And if you now save everything
and force a reload of the application
with this image added here,
you should also see a image here on the screen.
Though in my case, I got a typo here.
This should be assets/images.
The correct path of course, is important.
And with that I now see my dice here.
Now it's a bit big
and thankfully the asset constructor function allows
for additional configuration
by adding another argument to it.
In this case, the width argument, which you could,
for example, set to 200 pixels.
And with that, this looks much better.
And now we have a smaller dice image here.
So this is how you can display images on the screen,
and this is how you can work
with multiple constructor functions.

32. Adding Buttons & Using Functions As Values

Now, having this dice image here is nice,
but we, of course, also want to have a button below that,
which can be pressed to roll the dice.
And therefore, we should, of course, add such a button.
To do that, it's no longer enough
to just have a single child in this Center widget,
but instead, we need multiple widgets here.
Now, Center only has this child argument, though.

I mean, it has other arguments as well,
but those don't take any widgets at all.

So it only takes one single widget.

So what can we do now

if we actually wanna provide multiple widgets
as child widgets here?

Well, we need another widget in between the Column widget,
which is another key widget
that you will be using a lot when working with Flutter.

The column widget exists

to allow you to render multiple widgets,
multiple children above each other.

There also is a row widget,

which would do the same but next to each other horizontally.

Here I want to have widgets above each other vertically
and therefore, Column is the right choice.

Now, the children argument, which can be set here,
then takes a list.

So we need opening and closing square brackets,
and it takes a list of widgets.

So therefore, here, we can now take our image
and move that into that list here.

And then below that image, we can display our button.

Now, of course, buttons are a very common type of widget,
and indeed, Flutter supports three different main buttons.

It has the ElevatedButton here for displaying a button
that has a background color and a slight shadow.

It has the OutlinedButton for displaying a button
that has no background color, but a border.

And it has the TextButton for displaying a button
that is only some pressable text.

And that's the kind of button I want to display here,

though, of course, you can play around with that and also give other buttons a try. Now, when I use autocompletion here, so when I hit Tab after selecting TextButton, I automatically get two named arguments added here, child and onPressed.

Now, child is the simpler one.

Child simply wants another widget, which is wrapped inside of that button.

And typically, that's a Text widget which holds the text that should be shown on the button, because, of course, this TextButton should show some text; otherwise, you wouldn't see anything.

So here, this could, for example, be Roll Dice.

Now, I get some blue squiggly lines here because here we can add const to allow for some optional caching.

But what is this onPressed argument all about now?

Well, onPressed actually wants a function or null as a value, and that is something new.

Up to this point, we worked with a lot of different values; with text, with numbers, but also of course, with many objects and widgets, but we never used functions as values, but you can do that in Dart.

Functions in Dart are also just objects.

So if you define a function, you can pass a function as a value to arguments that want a function.

And here for onPressed, you have two main options.

Option 1 is that you define the function just in place.

This is done like this,
by adding opening and closing parentheses
and then opening and closing curly braces.
This is a so-called anonymous function.
It doesn't have a name like `clickHandler`
or anything like that, and it doesn't want a name.

Instead, this function
is defined here in this place
where a function is needed,
so that it's provided as a value to `onPressed`.

This function is then executed,
called internally by `TextButton`
when it registers a click on the button.

So this function is only used here,
can't be used anywhere else,
and that's why it doesn't have a name.

But between those curly braces,
you can define any code you want
that should be executed when this function is called.

So when this button is pressed.

That's Option 1.

Option 2 is that you define a method, for example, here,
after all our constructors and these variables,
and that could be a method called `rollDice`,
as you always did,
by providing a name and then also a return value, which,
in this case, should be `void`,
because `onPressed` wants a function that returns nothing.
So therefore, here, you should have `void`,
then your function name,
and then here, whatever code you want to execute.

And now you can use that function name

and use that as a argument value here.

So you don't execute the function here,

you don't add parentheses,

you just use the function name

to basically pass a pointer to that function

as a value to onPressed.

With that, onPressed knows where to find this function,

and internally, TextButton

will execute that function here when the button is pressed.

Now, I personally typically prefer this approach

of having a separate function that's then passed

as a pointer, so to say, by using the function name.

But you could also use this anonymous function instead.

But here, I'll go for the separate function.

And now here inside of Roll Dice,

we can do whatever we want to do

when this button is pressed.

If we save everything and reload,

we now see this button here.

It looks a bit strange.

It's at the top and the button styling isn't final,

but as you can see, it can be clicked.

Now it's time to fix the styling

and then do something when the button is clicked.

33. Styling Buttons & Working with Padding

So let's fix the styling of this here.

For that, I'd like to make sure that it's back in the center

and this button also should look differently.

Now, regarding the center part,

it's of course strange that it's not in the center

because it's still inside of that center widget.

But here it's important to understand that this column widget by default, takes all the available vertical space. It is indeed centered horizontally, as you can see. It's in the center regarding that. But vertically, column simply takes the entire available space, and therefore there, of course, is simply no way of centering it vertically. So in order to make sure that this is centered vertically, we must make sure that column does not take the entire vertical space. And this can be done with help of another argument that can be set on column. And that's the main axis size. The main axis for a column is simply the vertical axis. And here you got two main options. `MainAxisSize.min`, and `MainAxisSize.max`. These are two predefined values which you can use, and the default is max to take as much space as possible. Here, we therefore want min. And with that, if you save this, this is centered vertically. Now, for the styling of the button, we of course also wanna change this. And that can be done by adding the style argument to the text button. And actually you should move that in front of child, not because you technically need to, but because that is one of the patterns recommended by Flutter, to always keep your widget arguments last.

Now, style here wants a button style value.

And the easiest way of creating this for a text button is to use `TextButton.styleFrom`.

That's the end of function defined on this text button, which returns a button style object.

And `styleFrom` now takes various arguments, which you can use to overwrite the default styling.

For example, you can set the foreground color to change the default text color, because the foreground color of a text button is the text color in the end.

And here you could set this to `colors.white`, using this predefined color.

You can also add a separate text style, which then wants a text style object using this text style constructor function, which we already before used for styling the text widget.

So this constructor function.

And there we could also set the text color, but I'll just use it for setting the font size to 28 here.

And this can now also be const here.

And then with that, this text is a bit bigger and white.

Now I'd just like to have more spacing between the image and this button.

And for that, we have two main ways of doing that.

The first option is to add some padding on this text button, which you can do in this `styleFrom` function by setting padding.

Padding simply is some spacing inside of the widget between the widget content and the widget boundaries.

And this padding can be set here for this text button with the padding argument in the `styleFrom` function.

And it wants a `EdgeInsetsGeometry` object type.

Such an object can be created

with the `EdgeInsets` constructor functions,

which are suggested to me here.

For example, `EdgeInsets.all`

adds a certain amount of padding into all directions.

You can also add padding to just the top

by using the `only` constructor function instead.

And there passing the `top` argument, the named `top` argument,

which then must be set to a number,

like for example, 20 pixels.

This can also be turned into a constant value

by using `const` here.

And if you do this,

there is more spacing around this button.

Now the alternative approach

to using this padding setting here,

which is absolutely fine, I just wanna show you options,

is to comment this out, and instead add an extra widget

between the image and the button.

And that would be the `SizedBox` widget,

which kind of does what its name implies.

It simply creates a box of a certain size.

And you can add a child widget in there if you want to.

But most importantly,

you can also set a height and a width here.

And here we could set this to a height of 20,

and then also add `const`,

since this can be added for this widget.

And that would also add a widget of this size

between the image and the button,

hence adding this extra spacing here.

Now we don't see any widget here in between,
because this sized box doesn't have any content.
It's really just a widget of that size, of that height,
that just takes up that space
without showing anything on the screen.
And that's also an important difference
between sized box and other widgets,
like for example, container.
Container and most other widgets simply take the width
and height they need to take
to fit their content into themselves.
For example, a text widget is just as high and wide
as needed to show the text on the screen,
so that it's not cut off.
On the other hand, with sized box,
you explicitly set a certain height or width,
and that height or width is then set in stone.
If you would have some child widget inside of sized box,
that child widget and the content of it
would not lead to sized becoming wider or taller.
Sized box has a fixed width and height,
and if it would have some content
that doesn't fit into there
because it, for example, needs more width,
that content would be cut off.
And that's an alternative to adding padding.
You can use whichever approach you prefer.

34. How NOT To Build Interactive Widgets

So, now that we've got the styling right,
it's time to make sure that something happens
when this button is pressed.

For that, of course,
we have this rollDice() function which we connected,
but in there we're not doing anything.
Now, for a start,
let's simply say that in there,
we want to switch this image from dice-2
or whatever you used as a default image here
to, let's say, dice-4.
So, this should switch as we click this button.
To achieve this,
we could add a variable to this clause,
a real variable that can be changed
because now we do want to change things.
We could add a variable with the var keyword
and name this our activeDiceImage,
anything like that.
That could be our name here.
Now, of course, this also should have some initial value,
so we can set an initial value here.
We don't have to do this in an uninitialized way.
We can instead initialize this here
and for example, set this to this image path here.
That could be what we want to do as a start.
Now, with that, I can no longer add const here
because I brought back a variable.
And if we have a variable in there,
this is not a const class anymore.
We can't create constant objects with it anymore
that could be stored and reused from memory
because now, the object can change internally
because we have that variable which could change,
and which we indeed intend to change.

And we can ignore these blue squiggly lines
and this message here for the moment.
So, we have this variable,
and here, in rollDice(),
I then want to set activeDiceImage
to, let's say, the same path, but with dice-4.png.
So, that's not a goal.
We have this variable,
we're changing this variable here, then,
and we can then use this variable down here
instead of having this hardcoded path.
And this of course, also is a great example
showing us why we might want to use variables
in some situations,
because now we really do want
to have one of the same data container,
which we can reference here,
which will hold different values based on the circumstances.
This value here initially, and this value after clicking.
And if we save that,
and in main.dart, actually remove const here
because we had to remove const from GradientContainer,
we can add it in front of these Colors though,
but if we do that and make these changes
which we just discussed to GradientContainer,
save this and reload,
we got this.
But if I click this,
nothing happens.
So, maybe this function isn't executing, right?
To find out if it is,
we can add print here, a built-in command provided by Dart,

which we can use for outputting some information
in the debug console,
which is a console we as a developer can have a look at
whilst we're testing this app.

So here, we could output some text 'Changing image...'
something like this.

If we add this,
we just have to go to View, Appearance, Panel,
and here in this panel we find the DEBUG CONSOLE
which would print anything print outputs here
and some additional statistics
which are not important for us right now.

If I now restart this
and we go back and I click Roll Dice once,
I can see that Changing image...
was logged in that debug console.

So, clearly that code in a rollDice() executed.
Nonetheless, the image here didn't change.
Why?

35. Introducing Stateful Widgets

Now the image here did not change
despite the rollDice function being executed
because we're in a stateless widget,
and this is just one of two main ways of creating widgets
and it's the wrong approach for widgets
that should have internally changing data.
It is absolutely the kind of widget you should use
if you just have a widget that takes some input values
and then outputs some widgets.
But if you have a widget that has some data
that can change internally,

and where such data changes then should affect
the rendered UI, as it's the case here
where the image we wanna display on the screen in our own
widget here depends on this `activeDiceImage` variable.

If we have a scenario like this,
`StatelessWidget` is the wrong choice.

We should instead create this as a `StatefulWidget`.

So, what's the difference?

`Stateful` widgets, as the name implies, allow us
to manage state inside of them.

And state simply is data that may change over time
and that should impact the rendered UI.

So if the data changes, the user interface should change.

We simply have different states
of our user interface, you could say.

So therefore here indeed the first step is to convert this
to a `stateful` widget to make sure that in here
the data may change.

Though to be precise, it would be a bit overkill
to change the entire gradient container
to a `stateful` widget, since the majority of this widget
actually does not depend on that changing data,
it's really just this part here, this image, and the button.

Only these things work together to change the image.

The rest of the widget is pretty stable.

Therefore here it would actually be preferable
to break this widget up into two separate widgets.

One `stateless` widget, this one,
and one `stateful` widget which contains
the image and the button.

And that is what I'll do here
to keep the overall code organized.

For that I'll add a new file here,
which I'll name `dice_roller.dart`
because in there I want to create a widget
called `DiceRoller`, which will be that stateful widget
we're about to create.

Therefore `DiceRoller` should extend `StatefulWidget`.

And in order to do that we here also must import package
`Flutter`, and then `material.dart`.

So it's still the same import as before,
but now we extend `StatefulWidget` here.

Now this class here, `DiceRoller`, which extends
`StatefulWidget`, now is defined in a different way
than we did it before for the stateless widget.

Here we don't add a `build` method.

We do not do that if we have a stateful widget.

Instead in here we must add a `createState` method,
so a method called `createState`.

But just as with `build`, we should add
the `@override` annotation here
because that's basically another method we're forced
to implement because we're extending `StatefulWidget` here.

Now `createState` does not take a value here,
but it actually returns a value.

It returns a `State` object.

Now that's a type built into `Flutter`, provided by `Flutter`,
and `State`, like `List` earlier, is a generic value type.

Hence we have to add these angle brackets
to basically inform Dart which kind of state
will be managed here.

And here it should be the state for the `DiceRoller` class,
for the `DiceRoller` widget.

Therefore the value passed between those angle brackets

should be your class name, DiceRoller in my case.

And in the body of this create state method,

you then have to return such a state value,

so a value that has this type.

And this value is created with yet another class.

When using StatefulWidget, you will always work with two classes.

This is the first class here,

and then you must add a second class,

which typically should start with an underscore,

and I'll get back to that in a second.

And then DiceRollerState.

So the convention is to pick up your widget class name

and append State at the end,

and start with that underscore, that leading underscore.

This underscore has a special meaning in Dart.

It means that this class will be private,

it will only be usable in this file.

Even if you import this file into another file,

that other file will not be able

to access this State class here.

And the reason for that is that this state class

is really only meant to be used internally

by this DiceRoller widget class here.

Now this state class here also must extend something,

and that something is State again.

So this state value, though here again, we have to embrace

the fact that this is a generic type

and add the angle brackets.

And here in between, again, we add

our widget class name, DiceRoller.

So we extend this State class,

adjusted for our widget class name here,
and inside of this State class we now have to add
that good old build method again.

So stateful widgets are in the end split across two classes.
And the reason for that is simply that they're managed
in a special way internally by Flutter.

Flutter essentially requires these two classes
to be detached from each other.

Now the build method works just as the build method
in the stateful widget though.

You add `@override`.

You return a Widget, so a value of type Widget.

You will get this context parameter.

And then here you return your widget
or widget combination, your widget tree.

In my case here, that means that I want
to return this column here.

So I'll select this column, taking advantage
of these nice annotations which are shown in my code editor
here because of the Flutter extension that we installed
telling me where the column widget ends.

And I cut this from GradientContainer,
and instead return this here in `_DiceRollerState`,
adding a semicolon at the end.

And with that we have this separate state object.

Some things are missing here,
we'll take care about them in a second.

But now here in DiceRoller, in this createState function,
we now call `_DiceRollerState`,
with that leading underscore, as a function.

So we basically execute the constructor function
of `_DiceRollerState`.

We haven't added a constructor function here,
but as you learned, Dart automatically adds one by default
in case you don't add one yourself.

Speaking of constructor functions, we should add one here
to our widget class
to set that key parameter again.

So to accept this named key parameter
and forward it to the superclass, to `StatefulWidget`.

That's the same as for stateless widget.

And then we can also add `const` here.

We can do this even though a stateful widget by definition
is able to change internally

because we have this class separation,
and internally Flutter, in the end, makes sure
that the state object can change,

which is why we don't add a `const` constructor to that.

But the widget will be kept constant by Flutter.

It will take care of that for us.

So now we just need to make sure that this variable
and this `rollDice` function are there,

and therefore we can go back to `GradientContainer`,
and in there take this variable and this function
and cut it from there,

which then also allows us to bring back `const` here

because we removed that variable,

which also allows us in `main.dart` to bring back `const` here
and remove it in front of the colors.

But more importantly, we can go to `DiceRoller`,
to the `_DiceRollerState` here,

and there add both the variable and the `rollDice` function.

And we can get rid of the print statement here now.

With that we have the variable here and this function,

and therefore now we just have to make sure that we use the DiceRoller widget, which then is connected to that state inside of GradientContainer.

So there I'll now use DiceRoller and instantiate this like this.

And of course to make sure that it is available here we have to add an import because you always have to import what you use in a file.

So here I import from package:basics.

And then here dice_roller.dart.

If you save this and force a reload, you should now be able to click this button and change from a dice with two on it at the beginning to four.

Except for that it doesn't work.

So, what's missing?

I mean, we are using a stateful widget after all.

And indeed you must use StatefulWidget, but that alone is not enough.

Of course here we have a variable which we change, but even that is not enough.

Instead, by default, Flutter will simply ignore that.

You change your variable, that's good, but that does not lead Flutter to re-execute this build method and reevaluate the UI that it should output.

And it must re-execute the build method to pick up any changes to activeDiceImage because in here we have the code for displaying the image, and if activeDiceImage changed, the image on the screen will only be updated if the build method executes again, so that all this code is reevaluated.

Because otherwise if build never executes again,

it's still this initial UI snapshot that will be output that was based on the previous value of `activeDiceImage`. And that's a really important concept in programming in general actually, that you should understand. If you are using a variable like `activeDiceImage` here in your code, the code in which you use it, so in this case the build method in which we use it, does not magically re-execute just because you changed the variable value anywhere else in your code. This is not happening here, and this is not happening in other programming languages either. Instead, the value stored in this `activeDiceImage` variable is only considered and used if the code where it variable is being used executes. So for example, when this build method executes for the first time. And the build method will execute for the first time when this widget, this `DiceRoller` widget is rendered to the UI for the first time. And that, in turn, will be the case when the widget in which `DiceRoller` is used is being rendered, which ultimately all comes down to the `runApp` function in the `main.dart` file because that's where the entire UI widget tree starts being created. So that's when the build method here in `DiceRoller` executes in the end. And that's when Dart takes a look at `activeDiceImage`, and takes the currently stored path into account for, in this case, rendering an image on the screen. And initially, when this build method runs

for the first time because this widget is rendered for the first time, Dart therefore uses this dice-2 image, which is the initial value for `activeDiceImage`.

So if you then change that variable value, and you want Dart and Flutter to update the UI and take this latest variable value into account, you must find some way of getting Flutter to re-execute this build method of this widget, or of this widget's state in this case here.

So how do we get Flutter to re-execute build?

Well, we have to call a special function in the place where we update a value that is used inside of build, and that will lead to some different UI output.

We have to call the special `setState` function, which is available inside of State-based classes.

So inside of `_DiceRollerState`, since we extend Flutter's `State` class here, we have a `setState` function which we can call that is provided by this `State` class provided by Flutter.

We have to call `setState` here inside of `RollDice`.

And to `setState` you must pass a function.

And here you typically pass an anonymous function, as I mentioned it earlier, related to the button.

Inside of this anonymous function, you should now perform any updates to class variables, to object variables that should be reflected in the UI, because `setState`, this function which is provided by Flutter, in the end tells Flutter that it should re-execute the build function of this state in which you call `setState`.

And if, after executing build, Flutter determines that you now expect a different UI than what you had before,

it will go to the actual app UI and update it in all the places where updates are needed. So for example here, after setting `activeDiceImage` to a different path, when calling `setState`, Flutter will re-execute `build`. And indeed here, where we use `activeDiceImage`, it will now see that a different kind of image should be displayed on the screen, hence it will go to the real UI that is active on the screen and update the image in there. And with that, with `setState` added, if you now save this and force a reload, you will finally be able to click Roll Dice, and you see the image changed to a dice that shows us four. And that's how you build stateful widgets, and how you make sure that you can update the state, and such state changes are then also reflected in the user interface.

36. Generating Random Numbers

So we now added our `StatefulWidget`. And with that, we can click this button to change this dice here, the dice image. But of course, at the moment, we can only click it once. I mean, we can click it more often, but we always set it to this image that shows us a dice roll of four. And that's ultimately not the goal here. Instead we wanna generate a random value between one and six and update the image appropriately. Now, to get there, we'll have to do a couple of things, and most importantly, we have to generate

that random number.

Now that can be achieved

with help of a feature provided by Dart,

which we have to import though.

We have to add a import

where we import something directly from Dart

and there from the math sub package.

So this is now not a third party package

that would have to be registered in the pubspec.yaml file,

instead this is really from Dart itself,

but a Dart feature that is not available

in all Dart files automatically out of the box,

but which instead has to be unlocked in a file

by adding this import.

Here we want to use a math feature provided by Dart.

To be precise here in rollDice,

I now want to use the random constructor function,

which is provided by Dart's math package

to create an object that has various methods

that help us with the generation of random values,

random numbers.

Here, for example, the nextInt method.

nextInt can be called to generate a random integer.

And for that, nextInt actually wants an input, an argument,

which is the maximum, the highest possible integer

that should be generated.

Though that value, as we can read here,

will actually be excluded.

So if you pass in a value of 10, for example,

we would generate a random number,

which is greater or equal to zero and smaller than 10.

So here, if I want to generate a number between one and six,

I can pass in six
and that would give me a value between zero and five.
So to make sure that it's between one and six instead,
I simply have to add one here,
and that would then give me a number between one and six.
So that generates such a random value
and we could now use that random value
and store it in a variable called `diceRoll`, for example,
to then dynamically set this string here,
this path to an image.
So in the end, what I wanna do here
is I wanna replace this number
because my images are all named dice 1, dice 2, and so on.
So I just need to replace this number here with my number,
which is between one and six,
and it must be between one and six
because I only have images for one to six.
Now, getting such a number here into a string
is pretty easy with Dart
because it gives us a nice syntax, a nice feature
that allows us to kind of inject values into strings.
We can add a dollar sign here
followed by the variable name
that holds the value that we wanna inject into the string.
So now this will dynamically construct a path
which is `assets/images/dice-`
and then whichever number is stored inside of `diceRoll`.
That's what this special syntax does.
It injects the `diceRoll` value into this string.
So when we set active dice image to an image path,
this path will include the actual number
that was generated here.

That's one way of doing this.

However, we don't actually even have to store the entire path as state here.

Instead, we could get rid of that

active dice image variable here,

and instead just add a new variable

that could be named `CurrentDiceRoll`,

which initially maybe is two.

And then that's the variable I wanna set here in `setState`.

And I wanna set current dice roll here

equal to my `diceRoll` variable here,

or actually make this a bit shorter,

get rid of that extra variable

and just generate my random number here

and use it here to then update `currentDiceRoll`

inside of `setState`.

So now I just manage the dice roll number as state,

and that allows me to then grab this path,

which sets the dice roll dynamically,

or which injects that dice roll

and use that down here.

And instead of using `diceRoll` here,

I now want to use this `currentDiceRoll` variable,

which we do update inside of `setState`.

It's up to you which approach you prefer,

but this keeps our state a bit leaner,

a bit easier to understand

since we're just managing a number here.

And then we dynamically derive

the actual path based on that state value,

so based on that number.

Whenever `build` is executed,

and as you learned build will execute
when we call `setState` here.

With that, if we save everything and force a reload,
we should be able to click this
and generate different dice rolls.

Now of course, you might sometimes roll
the same number twice,
so simply click this a couple of times
to see some differences on the screen.

So that's our different dice rolls
showing up on the screen here
as we click this button.

But the current approach still isn't optimal.
Because at the moment, I'm always creating
a new random object here whenever `rollDice` is executed.
And whilst this works, it's a bit redundant.
Because technically this means
that every time `rollDice` is executed,
so every time we click the button here,
a new random object is created and stored in memory.
So in the memory of the device that is running our app.
And the old random object that was created there before
will be thrown away.

Now, thankfully it will be thrown away,
so therefore at least we don't occupy
the entire memory at some point.

But still that is some redundant
object creation and deletion in memory,
which is not a problem for this basic app,
but something you might wanna avoid.

And of course, my goal with this course is
not to teach you anyway,

but if possible, the best possible way
of solving a certain problem.
And therefore here it would be better
to cut this random object
and instead create a new final value here,
randomizer could be the name,
which stores this random object,
which we then use here.
And actually this doesn't even have to be
inside of our state class here.
This could even be defined outside of it,
maybe here right below the imports
because it's totally fine if this is only created once
when this code file executes for the first time
and it doesn't have to be re-executed
every time the state object is created.
Technically here, the state object shouldn't be recreated,
but still there's no need to keep the randomizer
inside of this class here in this application.
We can instead set it up
as a globally available final data container
here in this code file
by adding it right below the imports
and then still referencing it by name down there.
And with that, we should have the same result as before
that we can click this button and change the image,
but now it's a bit more efficient.

37. Module Summary

And that's it for this course section.
We built this roll a dice app here
where we can press a button to generate

a random number between one and six
and show different dice roll images on the screen.
And whilst we built this application from scratch,
starting with an empty main.dart file,
we of course had a very detailed look
at many core Dart and Flutter features you will need
for basically all Flutter apps you're about to build
and you will need as a foundation
for the rest of this course, of course.
Now as mentioned, it all started
with an empty main.dart file in this lb folder
in a new Flutter project.
Because it's this main.dart file that
is evaluated as the first file
when your Dart Flutter app starts on a device.
Now of course you, in this section,
learned that technically this Dart code
which you write in that main.dart file
and in the other Dart files is not executed like this
when the app starts on a device
but instead, after development,
when preparing it for publishing,
Dart parses your code files from top to bottom
and then compiles that code to native iOS
or Android machine code and it's that machine code
that's then shipped to those devices.
So that's then, in the end, installed
from the app stores, for example,
and it's therefore, that compiled code
that is, in the end, executed on those devices.
Nonetheless you, as a developer, write your code
with Dart and Flutter and therefore, you learned

that, indeed, it starts in the main.dart file
and there it starts with the main function.
And functions are one of the core concepts
you learned about in this course section
like many other programming languages,
Dart allows you the developer to create functions
which are simply custom commands
and you can, of course, not just create functions.
Instead, you can also use functions
that are built into Dart or provided by Flutter,
which is a Dart framework in the end,
and that's exactly what we're doing
in this custom main function here.
We're executing this runApp function,
which is provided by Flutter and which is why
you have to add this Flutter import to your main Dart file
to unlock these Flutter features
like runApp in this file here.
And it's then, in the end, the combination
of that main function in the main Dart file being executed
automatically by Dart and that runApp function
that's provided by Flutter telling Flutter
what should be drawn onto the screen,
so which kind of user interface
should be displayed on the screen.
And for that runApp needs a so-called argument,
it needs a widget or a combination of widgets
typically that, in the end, tells Flutter
what should be displayed on the screen
because Flutter, as you learned, is all about widgets.
It's all about combining different widgets in code
and that's also important.

In this section, you learned that you, in the end, tell Flutter what to display on the screen by using these so-called widget things in your code like MaterialApp, Scaffold, Center, Container, and many other widgets including custom widgets which we also built in this section, which you can use in conjunction to all these built-in widgets that are provided by Flutter. Like, for example, the buttons or various layout widgets that are provided by Flutter. In addition to those, you can also add your own widgets as you learned in this section.

And that is really important when building apps with Flutter, you have no visual drag and drop editor for doing that. Instead you build your UI in code.

You define the user interface by writing code and by combining these widget things.

And these things here are actually objects based on classes as you learned because Dart is an object-oriented language and therefore, all values in Dart are objects that includes simple values like text or numbers, but it also includes more complex values like widgets. Everything is an object, but those more complex objects are built based on classes which act as blueprints for objects.

And objects are, in the end, simply data structures stored in computer memory, so in the device memory on which your application's running.

And these data structures store data like, for example, the text of a text widget that should be output on the screen,

but also logic, functions that can be attached to classes like, for example, this rollDice function, which we added to a class in one of our custom widgets to make sure that when this button is pressed in our application, which is rendered with help of the text button widget provided by Flutter, that when that happens, when that button is pressed, this rollDice function is executed.

That's why we're using classes in Flutter, especially also when creating custom widgets as we're doing it here to group data and logic functions together into one data structure. And that was another core concept about which you learned in this section.

You also learned that Dart is not just object-oriented, but that types are important.

That Dart is a type-safe language, which simply means that all the values you are working with are of certain types, for example, the text 'hello World!' would be of type string.

A number could be of type int for integer if it's a number without decimal places and the MaterialApp widget would be of type MaterialApp because you can also create your own custom types.

And for example, every class you define also automatically creates a type of the same name.

You also learned that values are not just of one single type but instead, of multiple types.

They are always of type Object at least because all values are objects in Dart, but they might also be of other additional types.

For example, MaterialApp is not just

of the very specific `MaterialApp` type,
but it also has the more general widget type.
And this type feature exists in Dart to make sure
that you're not accidentally using
the wrong type of value in the wrong place.
Like, for example, passing some text as a value for width
when this actually wants a number.

Now, in this section when building custom widgets,
you also learned that there are two main kinds
of custom widgets: stateless and stateful widgets
with the difference being that stateless widgets
have no internal changing data which,
when changed, should update the UI.

Whereas stateful widgets indeed do support
internal data that may change and where such data changes
then should lead to an updated UI.

Like here our `DiceRoller` widget
which, in the end, allows us to change
the value of the `currentDiceRoll` variable
and which then makes sure that this change
and this updated value is reflected in the user interface
by calling this `build` method again which,
as you also learned, is a crucial part of custom widgets.

You add `build` methods to those widgets
to define which widget or widget combination
should be yielded by that custom widget
because custom widgets, in the end,
are about combining other widgets
and possibly enriching that combination
with extra logic as we're doing it here.

A key difference between stateless and stateful widgets then
is that for stateful widgets you actually,

create two classes, a widget class and a state class.

And behind the scenes Flutter makes sure that these two classes work together.

Whereas for stateless widgets you just have a single class which directly contains that build method, which I just explained.

And in this section you also learned how you can leverage constructor functions, which are functions that are executed when you create a class, when you use a class like we're doing it here with GradientContainer in the main Dart file when we create this widget combination to pass it to runApp that when you instantiate classes, the constructor function of that class is executed and this function can be used for some initialization work, but you can also use it to accept arguments, positional or named arguments.

And these parameter values can then be stored in internal variables, which then can be used in the build method, for example, so that you build reusable custom widgets where one and the same widget like GradientContainer here, could be used in different places of your application with different configurations and values.

And these were just the most important features about which you learned in this course section.

Of course, there also were other features covered like, for example, how the code editor,

Visual Studio Code in my case, can help with writing code.

You also learned about how const can be used to potentially optimize performance

and that, thankfully, Visual Studio Code tells you when you can use `const`. You learned about variables which are data containers and about final data containers, which are also like variables, but unlike variables can't be overwritten or changed. And therefore, at this point, you now don't just have your first basic app that you built together with me, but you also have that solid foundation, which you can now use to dive deeper into Flutter and Dart, which is exactly what we will do in the next course sections.

Flutter & Dart Basics II - Fundamentals Deep Dive [QUIZ APP]

Now that you have a solid Flutter and Dart foundation, it's time to dive deeper. And that's exactly what we're going to do in this course section here. In this course section here, we're going to build another app, a quiz app which allows you to start a quiz, then navigate various Flutter-related questions where the possible answers are actually displayed in random order. And once you went through all questions, you get a summary screen where you can see which questions were answered correctly and which ones were answered incorrectly, and you can of course also restart the quiz if you want to. And as you can see, this app is, of course, a bit more advanced than what we built

in the first course section,
which makes sense since we're diving deeper into Flutter.
And as we'll build this app
and go through this section here,
you will learn about additional, very-common Flutter widgets
that you should be aware of and how to use these widgets.
You will also learn about some new features
of existing widgets which we already did use
in the previous section.
You will learn how to render conditional and list content
and what exactly that is, what I mean by that.
But this will also be super important since you will need
these concepts in almost any Flutter app
you are going to build.
And overall,
since we're building this slightly more complex app here,
you get an overview and an understanding
of how such more complex user interfaces could be built
so that by the end of this section,
you will have a very solid Flutter toolbox
which you can use to build Flutter apps,
and which we then after this section, of course,
also will use to dive even deeper into Flutter.
But now for a start,
it's time to dive into this quiz app
and more crucial Flutter concepts.

38. Module Introduction

Now that you have a solid Flutter
and Dart foundation,
it's time to dive deeper.
And that's exactly what we're going to do

in this course section here.

In this course section here,
we're going to build another app,
a quiz app which allows you to start a quiz,
then navigate various Flutter-related questions
where the possible answers are actually displayed
in random order.

And once you went through all questions,
you get a summary screen
where you can see which questions were answered correctly
and which ones were answered incorrectly,
and you can of course also restart the quiz if you want to.

And as you can see, this app is, of course,
a bit more advanced than what we built
in the first course section,
which makes sense since we're diving deeper into Flutter.

And as we'll build this app
and go through this section here,
you will learn about additional, very-common Flutter widgets
that you should be aware of and how to use these widgets.

You will also learn about some new features
of existing widgets which we already did use
in the previous section.

You will learn how to render conditional and list content
and what exactly that is, what I mean by that.

But this will also be super important since you will need
these concepts in almost any Flutter app
you are going to build.

And overall,
since we're building this slightly more complex app here,
you get an overview and an understanding
of how such more complex user interfaces could be built

so that by the end of this section,
you will have a very solid Flutter toolbox
which you can use to build Flutter apps,
and which we then after this section, of course,
also will use to dive even deeper into Flutter.
But now for a start,
it's time to dive into this quiz app
and more crucial Flutter concepts.

39. A Challenge For You!

Now for this course section here,
I created a brand new Flutter project
with help of the Flutter create command.
For example, you could name it advanced basics
or anything like that.
And with that new project created in there,
you should go to your main.dart file in the lib folder
and clear all the content that's inside of it
so that you have an empty file again.
Because again, we're going to start from scratch here
so that we can really also practice
what we learned in the previous section,
because it is crucial that you got those basics down.
And attach to also find a zipped assets folder,
which you should unzip and add to your project,
which contains an images folder, which contains one image,
which we'll use as part of this section here.
Not immediately right now,
but a little bit later in this section.
And with that, you got this empty main.dart file.
And we're actually going to start
with a challenge for you here.

In order to practice what you learned,
I challenge you to build this starting screen of the app
on your own as an exercise.
And in the next lecture, we will build it together.
So you can use that next lecture
to compare your solution to mine.
But definitely try it on your own first,
write all the code that's needed
to bring something onto the screen
to tell Flutter that it should create some user interface.
Use the knowledge you gained in the previous core section
and try creating this user interface.
Now, one brief note or hint
regarding this user interface here.
This button here at the bottom is not a text button
as we used it in the previous section,
but some other kind of button.
And you could, for example, use the Flutter widget catalog
and there maybe you look into the material components
just as a hint to get an overview
of which buttons are supported in Flutter apps
and which button could be used here to create a button
that looks like the button you see here on this screen.
So as mentioned, my recommendation
is that you try this on your own
to practice what you learned.
In the next lecture,
we'll build this starting screen from scratch together.

40. Challenge Solution 1/2 - Creating a Widget

So did you succeed?

Were you able to build this starting screen?

In this video,

I will show you one possible solution

and I will show you how I would build this starting screen.

It all starts in this empty main.dart file of course.

And there you learn that you must add a main function

because this is kind of a special function

which is automatically detected and executed by dart

when your application starts on the target device.

This main function in the main.dart file

is automatically detected and executed.

And in this main function

you must call a special function provided by Flutter,

the runApp function,

which then will take a widget,

which then is the root widget

that will be drawn onto the screen

and which will then contain all the other widgets

as children and as descendants

to build the overall app user interface

as you learned it in the previous course section.

There you also learned that runApp

is not available in this file out of the box,

instead you have to import it from the Flutter framework

and you do this by adding the import keyword

and then the path to the file

or to the package file that you wanna import.

And here we indeed wanna import from the Flutter package,

hence we type package:flutter,

and then here it's the material.dart file

which is the main import file

when building Flutter apps.

You will import most of the Flutter features from this file.

So that's why we have to add it here
and with that we unlock this runApp function.
Now to this function,
you must pass a widget.
And typically here you start with the material app widget,
which is one of the most important
widgets provided by Flutter
since this will basically wrap your overall user interface
and do a lot of behind the scenes setup work.
So that's typically the widget with which you start.
And later in this course you will also learn about
the various named arguments that you can pass
to this widget constructor function
because there are a lot of things you can configure
about the overall app with help of this app widget.
But for the moment we just need to set the home argument,
the argument named home,
this named argument,
and we must set this to a widget
that should be displayed inside of our app
because the material app widget itself
doesn't really display anything,
instead it just provides a shell
with all that behind the scenes configuration work
as I mentioned
for the actual user interface we wanna display.
Well, and here inside of MaterialApp,
I'd like to set the Scaffold widget as my home widget
and that's another widget you saw
in the previous course section already
because where MaterialApp
sets up the general app shell so to say,

Scaffold can be used to set up the screen that will be displayed to the user.

For example,

you can use Scaffold to set a background color to the screen if you want to something we haven't done yet but which we will do later.

So that's why I'm using Scaffold instead of MaterialApp to configure some settings for the actual screen that's displayed in my app.

And here for Scaffold at the moment, I just want to set the body argument.

So a argument named body, accepted by Scaffold, and body just like home also wants a widget.

So here we also must pass a widget as a value.

And the widget I wanna pass here will be my start screen widget.

And for that I'll build a custom widget.

Now we could include this Scaffold widget into that start screen custom widget as well as I just mentioned,

that Scaffold does some screen specific setup work, so including it into the screen widget might make a lot of sense.

But since I plan on sharing the same background color and settings for all my screens in this app here, I will actually use this here as a wrapper around my different screens later.

And therefore I'll put my screen specific widget combination into a separate custom widget that will then be passed as a value for this body argument to Scaffold thereafter.

Hence, in this project in the lib folder,
I'll add a new file
because you learned in the previous section
that it's considered a good practice
to put custom widgets into their own files
so that's what I'll do here.
And I'll add a start_screen.dart file here.
Following that naming pattern,
that naming convention
you learned about in the previous section
where your file names should be all lowercase,
and if they happen to consist of multiple words,
these words should be separated by underscores
for better readability.
So here it's the start_screen.dart file which I add.
And in there we create a custom widget,
which as you learned is done with help of the class keyword
because we must create a class,
a blueprint for an object.
Because remember,
dart is an object oriented language,
all values are objects.
And for more complex values
like widgets or configuration objects or anything like that,
you use classes as blueprints
to basically tell dart
how these objects should then be created
and which kind of data and logic.
So functions should belong
to the object that will be created.
Since these objects are all about organizing data
and separating logic,

so that logic that works with certain data
is co-located with that data in the same object
and logic that does something else
is in a different object for example.

Therefore here we built our custom start screen widget
by using the class keyword
and then StartScreen sounds like a fitting widget name
though the name is up to you
but it of course should describe
what this widget is all about
and it should follow this naming scheme here
where you start with an uppercase character.

And if your widget name consists of multiple words
as this one here does,
then every word inside of the word
also starts with an uppercase character.

Now you also learned in the previous section
that your widget classes must extend
one of two main widget classes provided by Flutter
and that's either StatelessWidget or StatefulWidget.

And here for this starting screen,
since we at the moment don't need any changing data,
we can simply use StatelessWidget.

If you chose StatefulWidget
because we'll have a button in this widget,
that's not a wrong thought,
but we'll handle this button slightly differently later,
which is why you can use StatelessWidget.

Now to unlock this StatelessWidget class
from which we extend here,
we again have to import from Flutter,
from Flutter material to be precise.

And then inside of this custom widget class
we must add a build method.
That's what you have to add to your StatelessWidget.
The build method here must return a widget,
so a value of type widget,
and it should accept a context argument
which we didn't use thus far,
but which we will see in action
or which we will use later in the course.
We should accept it though
to make sure that we're implementing
this build method correctly
because StatelessWidget
actually wants us to implement build method
and it wants us to implement this method in a correct way
and we should therefore also add
the add override annotation here
to make it clear that we're in the end implementing,
and as it turns out also overriding
a method expected by StatelessWidget.
Now in the body of this build function,
this build method here,
we then have to return a widget
and for the moment to just have anything on the screen,
I'll return some text using the build and text widget
and in there I'll just say start screen like this.
Now we can add constant front of this
to unlock this extra performance optimization
where dart is able to reuse certain objects,
certain data structures in memory,
and Visual Studio Code simply tells us
that we can use const here

so that's quite convenient,
we basically can't miss this opportunity.
And we also have to add a constructor function here
by repeating the class name.
And in this function we should accept a named argument,
hence we need curly braces here called key.
And this argument needs to be forwarded
to the parent widget to StatelessWidget.
And as you learned in the previous section,
this can easily be done by using `super.key` here,
which is simply some dart shortcut for saying
that you wanna accept a argument named key
and forward it to the super clause
so the parent clause from which you're inheriting,
and there it should also be passed to an argument named key
and status widget indeed will accept a named key argument.
You can then also add `const` here in front of start screen
as you're also told by Visual Studio Code
so that you unlock this custom widget class
for being used in conjunction with `const`
so that it could then also be reused
after being created and stored in memory.
And with that I got a basic start screen widget class here.
Of course not the final version,
not the solution of this task but a start.
And back in `main.dart` file,
we can now use this
as a value for body in the Scaffold widget.
We can use the StartScreen widget here.
And in order to unlock this widget here,
we have to import it
because in dart,

whenever you are using something, for example,
a clause defined in some other file,
you explicitly have to import that other file,
dart will not automatically connect your files,
it does not do that.

So therefore here we add another import
and if you want to import from your own files,
you can still do this with help of package,
but now you use your project name in my case as basics.

And then there you point at the file
from which you want to import,
in this case start_screen.dart.

Now we can also add const here
to unlock these potential performance improvements.

And as you learned in the previous section,
you should add a comma after every closing parentheses here
so that if you then press the format document shortcut,
this code gets formatted
and is a bit easier to read and understand.

And with all of that,
we have a very basic widget tree using a custom widget,
and that's a great start
and we can now make sure
that we have an emulator up and running.

And with that run this app here on that emulator
and see whether we see our output here,
the start screen output on the screen.

And here that's looking good,
I have start screen here.

It's not centered.

You can barely see it,
but it's there.

I have this wide background from the Scaffold widget.

I have no error.

So this is looking good

and we can the offer now continue with this solution
to this first exercise.

41. Challenge Solution 2/2 - Working with More Widgets

So I got started with my solution here,

and to continue, I now want to change the background color.

Because in the expected solution that I showed you before,

I wanted to have a nice gradient background.

And that is, of course, also something

that we already implemented in the previous course section.

So in order to add this gradient,

we can wrap the StartScreen widget

with the Container widget,

because that container widget will then allow us

to set a decoration on it.

And I will indeed wrap the StartScreen widget

instead of putting the Container

into the StartScreen widget.

Since as I mentioned earlier,

I want to use the same screen setup and screen background

for all the screens I'm about to add.

And therefore, I will not put it into a specific screen,

but instead wrap it around this screen.

Now, to wrap a widget around my custom StartScreen widget,

I can right click on it and choose this refactor option,

and then wrap this with a Container,

so choose this option here.

With that, we add this Container widget

and we are now forced to remove const here

because Container does not support it,
at least not at the point of time I'm recording this.

But I can add it in front of StartScreen still

because that's inside of the Container.

And by custom StartScreen, which it does support const,
because of this const constructor.

And on Container, we now can add more options
than just child.

We can and should, for example, add the decoration option,
which allows us to add a BoxDecoration object
by using the BoxDecoration constructor function,
which then in turn can be used to set a gradient,
and that's exactly what I wanna do here.

And here I will again go for a linear gradient
as I did it in the previous section,
and format this code a little bit.

And to this linear gradient constructor function,
which will create such a linear
gradient configuration object, which then is used
by BoxDecoration and ultimately by Container,
we can pass a colors argument.

So a named argument with the name colors.

And colors wants a list of colors,
and you learned about this list type
in the previous course section as well.

It's a generic type, which simply means
that we got one type list in this case,
which works together with another type,
and it is quite flexible regarding which exact type it is,
here it's the type color,
but you could in other places of your app also have lists
of strings, or numbers, or any other values,

but here it's a list of colors.

And such a list is created by using square brackets.

And then between those square brackets,

you put your list values.

And that should at least be two colors

to create a nice gradient.

And to pick those colors,

I'll start with a nice deep purple maybe

like in the last course section.

And if you then hover over that,

you should get a nice color picker here

in Visual Studio Code.

And I will now choose...

Well, again, relatively dark purple color,

but a bit more purplish than in the last section.

And then copy that color

and add it again as a second value in that list.

But now, hover over that second color here

and choose a different purple tone like that.

Now, we can add `const` here

since all these configuration objects do have

`const` constructors and therefore, can use

this optimization technique here

by adding `const` in front of them when created as objects,

so when the constructor functions are called.

And then with that, if we save this

and we force an update on the emulator,

I do indeed have some purple here,

but that's, of course, not what I intended,

I'd like to have this on the entire screen instead.

Well, for that, we, first of all, have to pick some content

inside of the Container

that really occupies the entire screen.

And this Text widget at the moment does not do that,
instead that only occupies as much space
as is needed to display this text.

Now, that can be changed by wrapping it
with another widget, to be precise, the Center widget.

This Center widget, as its name implies,
centers its child widget, its content,
so in this case, this Text widget inside of it
both horizontally and vertically.

And in order to do that, the Center widget will try to get
as much space as it can.

And since this Center widget is in the end,
the first widget used inside of that Container
that's inside of Scaffold, it will in the end take up
the entire available screen because neither Scaffold
nor Container do by default take up any space themselves,
therefore, Center will basically expand
to take the entire screen width and height,
and hence, the surrounding Scaffold and Container widgets
will apply their settings and their decorations
to the Center widget.

That's how this here works.

And why using center here will make sure
that we get the background color on the entire screen.

So here I'll add a comma,
and move this const keyword here
in front of Center since Center also supports it.

And now with that, this looks much better,
now, this indeed does take up the entire screen.

Now, I'd like to have the gradient flow from the top left
to the bottom right corner

as I had it in the previous core section,
and for that back in main.dart,
we can go back to the linear gradient constructor function
and not just pass a colors argument,
but also a begin argument,
which wants an alignment geometry value,
which can be created by using alignment.
And then you could manually enter some coordinates here
or use one of these predefined values
by using the dot notation directly on the alignment clause.
And here I'll use topLeft as a begin value,
and for end, I'll use Alignment.bottomRight like this.
And with that, if I save this now,
this gradient indeed flows from the top left
to the bottom right corner.
So that's looking better.
Of course, that's still not all though,
inside of the StartScreen, I now want it
to have three main elements
if you take another look at the expected solution.
I wanted to have an image, then below that some text,
and then below that a button.
And when you have a scenario like this
where you want to have multiple widgets above each other
or next to each other, you might want to use Column
for widgets above each other
or Row for widgets next to each other.
Now, we'll see Row in action later in this section,
for the moment it's Column that we need
since I want to have multiple widgets
laid out on the screen vertically.
And Column wants a children argument,

an argument named children,
which then in turn wants a list of widgets.
So we create a list with square brackets,
and in there we identifying all the widgets
that should make up this column.
Now, Column does not support the const keyword,
hence you should remove it here to get rid of this error.
And now, we can get started adding our widgets here.
And I'll start with an image.
I did provide that image earlier in this section already,
it's this assets zip file,
which you found attached to lecture 2 in this lecture.
In that Assets folder,
you will find a sub folder named Images.
And in that Images sub folder,
you have this quiz-logo.png file.
And it's now this image which I want to output here.
And to output such an image, it's not enough
to just add it to your project,
instead as you learned in the previous course section,
you also have to go to your pubspec.yaml file.
And in there, if you scroll down,
you should find this assets section,
which you should comment in so that it is active.
And then there below assets
after hitting the Tab key once to indent this,
you should add a dash, and then basically list
all your assets, all your images, for example,
that should be usable in this Flutter app.
And for that, you specify the relative project path
to that asset.
So in my case, assets/images/

and then the name of the image.

And once you did that, this asset can be used in your code.

And you can use this image asset

and display it with help of the Image widget,

which is also built into Flutter.

The Image widget can be constructed like this

and then you would have to provide an image argument,

which wants an image provider.

But if you have an asset, a locally stored image

that should be displayed, it's easier to display that

by using another constructor function

offered by the Image widget class,

and that would be the asset constructor function.

To that, you simply pass a string

created with these single quotes

that contains the path to the image

that should be displayed.

So in this case, assets/images/quiz-logo.png.

And you're good to go.

If you now add a comma here for better formatting

and you then save everything,

this image should be displayed on the screen.

At the moment, it's a little bit too big though,

I'd like to shrink it and I'd like to center it vertically.

Now, to shrink it, you can pass another named argument

to this image asset constructor function,

and that would be the width argument.

And here you can set a width in pixels, for example, 300.

If you do that, the image will be shrunk.

Now, to center it, since we are already here

in a Center widget, you can simply go to your Column,

and there set the mainAxisAlignment to MainAxisAlignment.min,

so that instead of taking all the available vertical space,
which it will do by default,
the Column which it will only take
as minimal vertical space as needed,
so only the height of this image in this case.

And `mainAxisSize` refers to the vertical size,
because for a Column, the `mainAxis` is the vertical axis.

So with that, this image is now in the center.

Of course, this image was just one of the widgets
that should be added here inside of this Column.

And it's therefore, time to also add
the two remaining widgets.

Hence, next I wanna add some text,
since some text should be displayed below the image.

And here the text that should be displayed is,
"Learn Flutter the fun way."

Like this, and here you can add `const` in front of it
since this text, which it can be optimized.

Now, if I do that, this text indeed does show up here,
but it got a couple of problems.

For one, it's way too close to the image.

It would be better to have some spacing
between image and text.

And, of course, I also want this text to be a bit bigger
and to have a white color or some color
that's a bit easier to see
in front of this purple background.

Now, for the spacing, you can, for example, add a `SizedBox`
as an extra widget between the image and this Text widget.

`SizedBox` is simply a widget where you, as a developer,
can define the exact size of the widget
so that it does not take all the available space

like Center does, or just as much space
as the content needs as Text does,
but that it instead takes exactly that amount of space
that you tell it to take.

And here we could, for example, set the height argument,
so pass a value for the named height argument
of let's say 80.

You can then also add const in front of this.

And with that, we got some spacing here, which is good.

Now, of course, I also wanna change
the text color and size, and that can be done
by going to the Text widget and adding this style argument.

This, as you learned in the previous section,
should be set to a value of type text style
and such a value can be created
with the text style constructor function
that is provided by Flutter.

So we're using a built-in class here, provided by Flutter.

And to this text style constructor function,
you can, for example, pass a color value
and set this to white as a color.

Or maybe if you want to get fancy,
pick a color that is maybe some very light whiteish purple,
something like this, we can give this a try as well.

I also wanted to increase the font size,
and for this we can use the fontSize argument here
and set this maybe to 24.

If you do that and save everything,
the text is now bigger and easier to read.

Now, last, but not least, I wanna have a button
below that text that can be clicked
to then later start the quiz.

And here in the expected solution,

I actually wanted to have a button
that has a visible border.

So for that, I'll first of all, add a comma here
and format this to make this easier to read.

But then below this Text widget,

I'll add another SizedBox widget to add some extra space
between the text and the button.

And this can be const and can, again, have a height
of let's say 30, let's see if this works.

And below this, we can then add the button.

And for adding a button that has a visible border,
we can go to the widget catalog provided by Flutter.

And there under Material Components, we find a Buttons area.

And in that area, we, for example, find
this outlined button, which looks like the kind of button
I want to use here.

So to use it, we can then create it here, OutlinedButton,
and to the onPressed argument,

I'll simply pass an empty anonymous function right now.

You learned that you can create anonymous functions
like this in the last section as well.

So this is how you can create a function
that is only available here in this place
and that's passed as a value to the onPressed argument.

But since it contains no code in the function body,
this function won't do anything here.

Of course, we're going to change this later,

but first, I want to go to my child here

and set this to a Text widget so that we can define the text
that should be shown on this button.

And here I'll just add a text of, "Start Quiz."

And this can be turned into a constant object
by adding the `const` keyword.

Now, with that, we have that button here,
but I'm not too happy with the styling.

So to change the styling, we can add a style argument here,
and this should be set to a button style.

And whilst you can construct a button style manually
with the `ButtonStyle` class provided by Flutter,
the easier way of doing that, which you also kind of saw
in the last section already, there just with `TextButton`
instead of `OutlinedButton`.

But the easier way, which works for `OutlinedButton` as well
is to use the `OutlinedButton` class,
and they are called the built-in `styleFrom` method.

And `styleFrom` then makes it a bit easier
to apply various style settings to this button.

For example, here we can set the foreground color,
which will be the text color in the end,
and set this for example to white.

And with that, the button looks like this now
and the text is way easier to read.

And that's the final solution for this exercise.

That's where we're going to continue in the next lecture
to learn more about Flutter, available widgets,
available configuration options, and much more.

But this is the solution to the exercise,
and this also was a quick summary of some core concepts
and widgets about which you learned
in the previous course section.

Therefore, it's now time to continue
and learn some new things.

42. Adding Icons to Buttons

So let's continue working on this app.

And of course, an important next step

will be to make this button work

so that we then actually visit a different page

on the screen where we get a question,

which we then can answer.

But before we do that, I wanna tweak this button a bit more,

the styling or the look of this button, to be precise.

And I also wanna tweak this image a little bit.

Now, for the image, I wanna make it slightly transparent.

For the button, I want to add an icon inside of it

so that it's not just some text here,

but instead I'm text combined with an icon.

And we'll start with that,

with that button where I want to add an icon,

because that of course could be a very common requirement

in the apps you want to build.

And thankfully,

it is quite simple and straightforward to do.

If you wanna turn this outline button here

into an outline button with the icon,

you can use an alternative constructor function

that's offered by Flutter on this widget.

For this, you can add a dot

to access this extra constructor function.

And here it's the icon constructor function.

And this does what the name implies.

It adds an icon to the button.

Now if you use that constructive function,

you can still use `onPressed`, and you can still use `style`,

but `child` is no longer called `child`.

But instead here, it's the label argument
that should be used to define the widget
that then defines the non-icon content of this button.

But of course, you also do want to add an icon,
and that's done with the icon argument
that can now be set when using
this icon constructor function.

And this icon argument here wants a widget,
and therefore technically,
you could of course provide any kind of widget here,
for example, also another text widget.

But typically, you of course indeed want to,
well, create an icon here.

And for that, Flutter gives you the icon widget,
which, and that's really important,
can also be used outside of this icon button.

Icon is a regular widget provided by Flutter,
and you can use it anywhere else where widgets can be used.

For example, here instead of this column widget,
if you wanted to add a standalone icon.

If it should be a clickable icon though,
you want to use something like outline button icon.

And then here you set the icon argument
to such a icon widget.

Now this icon widget constructor function
then needs some icon data.

It actually takes a bunch of named parameters
that can be set,
but it has one positional argument that should be set.

And that's the icon data, which is simply a pointer,
you could say adds the concrete icon
that should be displayed on the screen.

And for this, Flutter gives you the utility icons class,
which when used with a dot here,
offers you a bunch of predefined icons.
And if you cycle through these icons,
you should actually also get a preview of these icons,
which makes it very easy to pick
just the right one for your use case.
And here I want to use the arrow right alt icon here,
but of course you can use any icon you want,
but this icons class here makes it very easy to use
all these built-in predefined icons that ship with Flutter.
And with that, we just have to add const here,
or we should add const here, to be precise.
And if you now save everything,
you should have this button with this icon here now.

43. Adding Transparency to Widgets

So that's how icons can be added.
As I mentioned, I also wanna tweak this image here
a little bit though,
and I wanna make it slightly transparent.
Now, there are two main ways of doing that,
and whilst one approach is better than the other,
I still want to show you both here.
The first approach, which is typically not recommended,
but which you should still know,
is that you wrap this image widget with another widget.
Now, not a widget, which is suggested here,
but instead we can use this generic wrap
with widget refactoring action here,
and then we can enter the widget name here manually.
In this case, it's the opacity widget, which I want to use.

Now, the opacity widget then does not just need a named child argument, but instead it also needs an opacity argument. And indeed, this opacity argument is required. It must be said when using this Opacity widget. And opacity now takes a number value between 0.0 and 1.0, where 1.0 is fully opaque, so fully visible, and 0.0 is fully transparent, so fully invisible. So if I set this to 0.5 here, for example, you'll see that now this image has some transparency. If I set it to 0.1, this is even clearer to see because now it's almost invisible. So that's one way of setting the transparency of a widget, of the widget that is wrapped by transparency to be precise. Now, this can be useful, but behind the scenes, this is quite performance-intensive. Not a huge problem for this basic app here, but typically you should avoid using opacity if you can avoid it. And that's why I'll copy this and then comment it out and re-add it, but now I'll remove this Opacity widget here and go back to just having the Image.asset here and show you an alternative approach. Because whenever you have a widget where you can set a color, there is a better approach available, a better approach than using opacity. And it turns out that for image, you can set a color argument when using this asset constructor function, and this color can be used

to apply some transparency to this image.

Now for a start, let's set this to `colors.red`,

so to the standard built-in red color,

and let's see what that does.

And as you see, it changes the color of the image,

and that's also good to know.

This color argument on the image widget can be used

to add an overlay color on top of the image,

which sometimes can be exactly the effect you want,

but this gets even more powerful

if you now apply a transparent color here.

And for that here, since I want to keep my image white,

I'll first of all switch to white

and then here we have this transparency slider.

And here it's fully opaque if this is at the top.

Here at the bottom, it would be invisible.

And we can now tweak this and maybe go somewhere here

so that this first value here is 150.

And with this, we now add a white overlay color

that has some transparency applied to it.

Now this can be set to a const value

by adding the `const` keyword.

And if you do that now, you got a white image again,

but a slightly transparent image.

And of course here, you can also play around with that.

If you set this to 20, for example,

it's almost entirely invisible.

If you would set it to 255, which is the maximum value

that is allowed here, it would be fully opaque.

And here I'll go for 150, which is what I had before

to have a slightly transparent image.

And these are some additional styling things

that you can do in your app
to make it look exactly the way it should look.
So for the moment,
that's now my finished start screen widget
with the image here with some text
and with this outlined button.
The next step now is to make sure
that this button actually does something when we press it,
that it does lead us to a different screen.

44. Repetition & Exercise: Adding a Stateful Widget

So, let's now work towards the goal
of making this button work so that when we tap it,
we navigate to a different screen
that shows us these questions which we can answer.
Now for that, we, of course, need
to tweak this function which is passed as a value
to onPressed, because in the end, that is the function,
and therefore this here is the code
that will be executed when the button is pressed.
But what exactly should we do here?
Well, in the end, what we wanna do here
is we wanna replace this StartScreen widget here
with a different widget so that instead
of showing StartScreen inside of our scaffold
and container here,
we show that questions screen or however we want to name it.
So, for that, of course, we need a new screen,
but then we also need a way of replacing
this StartScreen here with that other screen
after this button down here was pressed,
so after this function here executed.

And for that to implement this solution,
we'll need to dive into a couple of new concepts.
To be precise, over the next lectures,
you will learn how to render content conditionally
and how to lift state up.
Now, before we start rendering content conditionally
and lifting state up,
I'll first of all create two new widgets.
I'll create a new widget, which, in the end,
will contain my MaterialApp here with all its content
because we'll need that custom widget a little bit later
for lifting that state up,
but I'll get back to that once we're ready for that.
I just wanna create the widget for this already.
And we'll need another new widget for this questions screen
to which we wanna switch.
So, for that, in the lib folder,
I'll add a new file called quiz.dart,
which should contain this MaterialApp here,
and another new file
which could be called questions_screen.dart.
And in both files, new widgets should be created.
And, of course, to practice what you learned,
definitely try building these two widgets on your own first.
It should both be stateful widgets
because we will later need these widgets to be stateful.
And in the previous course section,
you learned how to build stateful widgets,
therefore definitely try this on your own.
In the questions_screen widget,
you can then simply return some text widget
or anything like this.

And in the quiz widget,
you should return this MaterialApp with all its content.
You don't need to do anything else with these two widgets.
You don't need to import them or use them in other files.
Just create them.
And after a short pause,
which I give you to pause the video
and try creating these widgets on your own,
I will build them together with you
so that you can compare your solution to mine.
So, did you succeed?
Let's now build these two widgets.
And I'll start with the quiz widget in the quiz.dart file.
For that, we again use the class keyword,
and then I'll name it Quiz,
not QuizScreen or anything like that, just Quiz
because that will basically be my main widget
that later manages the entire quiz.
And as mentioned, this should be a stateful widget,
hence you should extend StatefulWidget,
which is a class provided by Flutter,
and therefore, of course,
you must import package:flutter/material.dart
as you learned it.
With that, we create this stateful widget.
And as you learned in the previous section,
when building a stateful widget,
you in the end always end up
with two classes that must be built.
So, therefore, besides building
this StatefulWidget class here,
we also need a second class,

and that will be the `_QuizState` class.

The name is generally up to you,

but the convention is to use your widget class name

and append state,

and also add a leading underscore

to turn this into a private class,

which technically can only be used in this file here,

in the `quiz.dart` file in this case,

and not in any other files,

because this class, this `_QuizState` class,

will be connected to this `Quiz` widget class here soon.

So, that's why you should name it like this.

And this also extends something provided by Flutter,

and that something is the `State` class,

which is a generic class, though,

as you can tell by these angled brackets.

And in the end, that simply means that this `State` class

and type is closely connected

to another class and type therefore.

And to kind of establish this connection,

you add these opening and closing angled brackets

as you learned it.

And between those angled brackets,

you have the name of that other type

to which this class will be connected in the end.

And here, that's the `Quiz` widget class.

So, you add this here.

And inside of this `_QuizState` class here,

you then must add a `build` method which returns a widget,

which takes this context argument,

and which should also receive this `@override` annotation.

And then in here,

you return your widget or a combination of widgets,
and as mentioned before for this Quiz widget,
that should simply be the MaterialApp here.

Like this, so I'll cut this here
and then go to my Quiz widget file
and return MaterialApp here.

Like this.

Now, in there, I'm using this StartScreen widget,
and therefore, I have to bring this import here
from the main.dart file into my Quiz widget file here.

Like that.

Now, with that, we're not done, though.

That's just the _QuizState.

That's not everything we have to do.

Instead, we now have to go to the Quiz widget class here
and also add something to that class.

And here, you should add a createState() method,
which must return a State value,
and State, as mentioned, is a generic type,
so here, it must also be defined in a generic way.

And here, you also refer to that Quiz widget class,
which, in the end, just says

that createState() creates an instance of a class
that's based on State connected to Quiz.

So, an instance of this _QuizState class here in our case.

And that's therefore what you should do,
inside of createState(),

you should return an instance of this _QuizState class
by calling the _QuizState class constructor.

Of course, here, we haven't added a constructor function,
but as you learned in the previous section,
you also don't have to add one.

Instead, Dart automatically gives you one if you haven't added a custom one.

Now, here, you should then also add `@override` on `createState()`.

And for the Quiz widget here, we also then should add a constructor function where you should accept and forward that named key argument and turn this into a const constructor.

And that is now this Quiz StatefulWidget.

That's the widget class, and then below that, we have the `_QuizState` class, which, in its build method, returns this MaterialApp.

So, that's the first of the two widgets that should be added.

The second widget should be added in the `questions_screen` file here, and they are just as before, you add your class `QuestionsScreen`, which extends `StatefulWidget`.

And, of course, here you then also import from `package:flutter`, and then there, `material.dart`.

And we add our `createState()` method here.

And to do this a bit quicker, you can also open this suggestions menu by pressing `Ctrl + Space`.

And with that, `createState()` suggestion's selected.

If you hit `Tab`, this is added for you, and you just need to replace `StatefulWidget` here with `QuestionsScreen`, and replace this part inside of `createState()` with the actual code where you create a state.

And for that, we need to add that second class, of course,

which in this case, is called `_QuestionsScreenState`,
and which extends `State` connected to `QuestionsScreen`.
Like this.

And then, of course, inside of that `State` class here,
you add the `build` method,
and for that, you can also start typing `build`
and then choose this `build` suggestion.

Hit `Tab`, and it's added for you.

You can get rid of this `type` annotation in front of `context`.

It's actually not needed here.

And then inside of this `build` method,

you just return a widget,

for example, the `Text` widget

where you could say `'Questions Screen.'`

Like this.

And, of course, we'll change this later,

but it's some good dummy content for now.

Inside of `createState()`, you then return an instance
of this `_QuestionsScreen State` class like this.

So, you instantiate this class,

you create an object based on this class, so to say.

And, of course, just as before,

you should add this constructor function

where you get and forward that `key` argument.

And that's these two classes added.

Now, with them added,

I at least wanna use the `Quiz` widget which we added
in the `main.dart` file,

because here, I want to output my `Quiz` widget

or pass this to `runApp`.

And I can do that since inside of that `Quiz` widget,

I am, of course, using `MaterialApp`.

So, in the end, by using my custom widget here and passing it to `runApp`, I still use this `MaterialApp` just wrapped into my custom widget here.

And for this, for this Quiz widget to be available here, of course, you also must add a `import` statement to this `main.dart` file and `import` from your own package here, and they are from the `quiz.dart` file like this.

You can then add `const` here because Quiz does have a constant constructor.

And if this is now saved and you force the app to reload, you should have the same output as before, but now the app is prepared such that we can start working on this button.

45. Rendering Content Conditionally

Now as mentioned before, to make sure that we switch to a different screen, if we tap this button, we must dive into two important new concepts, Rendering Content Conditionally and lifting State Up.

And I wanna start with a Rendering Content Conditionally.

In our quiz widget, which is the main widget we use for starting our overall user interface, we are at the moment displaying the `StartScreen` in the end.

Wrapped inside of material apps scaffold and our container with a gradient, but wrapped in there, we're outputting the `StartScreen`.

And if you wanna replace some content on the screen with some other content, for example, after a button has been pressed, you simply have to replace the widget here in your code

because the entire user interface is controlled with code.

So what we could do here is that in our QuizState class, maybe, we have a variable,

which could be the activeScreen, if you wanna call it like this.

And activeScreen could be set equal to StartScreen like that.

And this is something we haven't done up to this point, but you can store widgets in variables.

This is absolutely allowed because widgets are just objects and objects are just regular values in Dart.

So you're not limited to storing some text or some numbers like 30,

but you can store widgets inside of variables and that's what I'm doing here, storing a widget in a variable.

And of course here we can add const, but it's still a widget.

And now we can of course add a method here to this QuizState class,

which could be called switchScreen, where we want to navigate to a different screen.

So which returns no value, but in which we use setState, a function provided by Flutter that's available inside of classes that extends State, as you learned.

And to setState, you have to pass a function, for example, an anonymous function.

And in this function you then should perform any data changing activities that should be reflected on the user interface.

So in there we could set `activeScreen` to our `QuestionsScreen` here.

And for that of course, `QuestionsScreen` must be imported.

So you must add a import statement where you import from your own package.

And there from the `questions_screen.dart` file.

With that, `QuestionsScreen` should be available if I add the missing `S`.

And we would now override the value stored in `activeScreen` with this screen widget here, when executing `SwitchScreen`.

Now at the moment this causes an error because a value of type `QuestionsScreen` indeed can't be assigned to a variable of type `StartScreen`.

And we get this error because `activeScreen` infers its type from the value that's used for initializing this variable.

And this value whilst being a widget is of course also of type `StartScreen`.

Because whenever you create a class in Dart, that class also automatically gives you a type of the same name.

So the `StartScreen` class, which we defined here also gives us a `StartScreen` type.

And that type in the end is based on the widget type, but is a bit more specific than the widget type.

It's specific to this class with exactly its details, its extra properties or whatever it might have.

So in order to make sure that `activeScreen` is a bit less restrictive and accepts all kinds of widgets, not just `StartScreen`, you can replace `var` here with `Widget`.

And that's one of the use cases
or scenarios where you indeed would want to replace var
with a more specific type.

Because here I don't want to use start type inference
because it's too restrictive.

And instead set activeScreen to a more general type,
which in the end is met by both StartScreen
and by QuestionsScreen.

And now if we add const here as well,
we would have some valid code
that would actually change this variable value
whenever this function is executed.

And how does this help us?

Well, this helps us
because now we can use the activeScreen variable down here
as a value for a child because child,
and that's important to keep in mind, just wants a widget.

And of course activeScreen now stores a widget.

So we can use activeScreen as a value here for a child
because this simply leads to Dart looking into this variable
and using that value
that's currently stored in this activeScreen variable
as a value for this child argument.

So that's how we could conveniently output different widgets
here by simply storing different widgets in a variable.

And since we call setState,
and that's also why it's important to call setState,
this build method will be executed again
because that's what setState does in Flutter apps.

When you call setState in a state class,
so a class connected to a stateful widget,
Flutter will re-execute the build method

and therefore run all this code here again.

And then it will compare the result of executing that code

to the result it got the last time

it executed the build method.

And if there are any differences, it goes ahead

and updates the rendered UI accordingly.

And indeed here for this widget, for the quiz widget,

when the build method is executed for the first time,

activeScreen will be set to StartScreen and that will

therefore be used as a value down there

and hence the StartScreen would be displayed.

But if then for whatever reason SwitchScreen executed

and activeScreen would be set to QuestionsScreen with help

of setState, the build method would run again.

But now activeScreen is equal to QuestionsScreen

and therefore down here QuestionsScreen would be output.

And Dart would detect that there are differences

between these two widget trees, the old one and the new one,

and it would update the UI accordingly.

That's important to understand,

which is why I am emphasizing it.

And this approach we're using here

where we're setting a variable

to different values is one way

of rendering content conditionally.

It's not the only way

and you'll see some other ways later as well,

but this is one way,

but of course at the moment we're never

executing SwitchScreen.

This SwitchScreen function here,

the SwitchScreen method is not connected to this button.

And that's of course what must change.

And that's exactly what we'll do by lifting the state up.

46. Accepting & Passing Functions as Values

So what's this lifting state up concept
all about?

Well, at the moment in our application we have the problem
that we have the StartScreen, which welcomes the user
and provides the Start Quiz button.

But we then also have the QuestionsScreen
to which we wanna navigate when that button is pressed.

So the problem that we have here is that the button
that should trigger the state change is in one widget,
and the widget that should be affected by the state change
is another widget.

So we have kind of two widgets that need to work
with the same state or that depend on the same state.

And the solution here is to lift the state up,
which is why we already created that quiz widget.

If we manage our state in that quiz widget,
and we then also have the function
for manipulating the state in there,
we can conveniently move between these two different widgets
by rendering content conditionally.

And that's of course what we're already doing here.

We do already manage and update some state
in this quiz widget in the end,
in the state class that is connected to this quiz widget.

And we are rendering the content conditionally already.

So we basically already lifted the state up,
but we haven't connected this switchScreen function
to the screen that contains the button

that should trigger this function or this method here.

And that is another important,

and therefore, at the moment, missing aspect

of this lifting the state up concept.

It's not enough to just manage the state

in some shared parent widget, Which quiz here is,

it's the parent widget

of both StartScreen and QuestionsScreen,

but it's not enough to just manage the state here,

but instead you also must make sure that the child widget

that should initiate the state change,

in this case, the StartScreen widget,

gets access to the state changing function

defined in this parent widget.

So in this case, StartScreen needs to get access

to this switchScreen function.

And how do we now do that?

Well, you might remember that in the previous course section

I did explain that functions in Dart

are also just regular values.

There we also had a button

that should trigger some function,

and in that case back then it was a function,

a method to find in the same file,

but still it was a method

that should be triggered when a button was pressed.

And to do, that we passed a pointer at that function.

So we used the function name in the end

as a value to that button,

and internally that button would then call that function

when the button was pressed.

And we can do the same here with our own custom widget.

We can pass a pointer at switchScreen to StartScreen.

So simply by using the name of that method,
switchScreen, like this.

So you don't add parentheses here
because that would execute this function
when this line of code here is executed,
and that would be too early.

Instead you just use the name
of the function or method here,
and that in the end, under the hood, passes a pointer
to that function as a value to StartScreen.

And inside of that StartScreen widget,
you can then use that function to call it whenever you want,
for example, when some timer expired
or when this button down there was pressed.

Now, at the moment, this doesn't work
because we're basically passing in too many arguments
to StartScreen because I'm passing in one argument here.

The pointer at the switchScreen function,
but StartScreen at the moment doesn't want
any positional arguments.

It just has this one named argument here.

Now, that can be changed though.

We can add a positional argument to StartScreen,
and we could also add a named argument,
but here I'll go for a positional argument,
and this argument could be called startQuiz.

The name is totally up to you.

But now we're accepting this positional argument,
and we should also define the type of this argument
by adding it in front of the argument name.

And here, the type is Function,

which is a special type built into Dart.

To be precise, it's a void function,

and you should now also add parentheses here,

which simply means that startQuiz should be a function,

and function is a type built into Dart.

So it should be a function that takes no arguments,

because indeed, switchScreen does not take any arguments.

So as a value for startQuiz,

we want to get a pointer and a function

that takes no arguments

and that also doesn't return a value.

And that's exactly what switchScreen does.

It is a function that takes no arguments

and returns nothing.

And with that we're passing a pointer at this function

as a value to StartScreen.

So now we can call this function from inside StartScreen

by simply executing startQuiz like a function,

because it is a function basically,

startQuiz is an argument

that contains a function as a value.

That's what we set up here.

And I'm spending a lot of time on this

because this can be a super confusing concept,

but in the end, it's all about a standard Dart feature.

You are able to use functions as values.

and therefore to also pass them as values to other functions

or to constructive functions as we're doing it here.

But with that alone, we're not done.

We're accepting this argument here.

But as you also learned in the previous course section,

this alone is not enough,

because the arguments you accept here
in your constructor function
can, for example, not be used in your build method.
So we can't use it down there where we would want to use it
because that's where our button is
to which this function should be connected in the end.
What we should therefore do is we should add a new variable,
or even better, a final variable here to StartScreen,
because we'll only set this once
when the StartScreen widget is created.
And this final variable will be of this type,
so we can cut this type,
and here we could name this startQuiz as well.
And now with that name being equal to that name,
we can use that shortcut built into Dart
where we add this dot in front of startQuiz
to automatically set the value we get here
for this argument as a value for this variable here.
So in the end, when this widget class is created,
when the object is instantiated,
this startQuiz variable is set
to the value we receive on this startQuiz argument.
And with that we now have this variable,
this startQuiz variable,
which can now be used anywhere in this class
including this build method.
So now down here for onPressed on this Outlined Button,
we can use startQuiz and execute it like a function
in this anonymous function.
And by the way, here, we're doing the same thing
as we did here in the QuizState class.
Here in QuizState,

we're passing a pointer at switchScreen
as a value to StartScreen.
In the StartScreen widget,
we're passing this anonymous function,
a pointer at this anonymous function therefore,
as a value to onPressed,
that's the same pattern in the end.
And in this anonymous function body,
we're executing our own startQuiz function.
But this anonymous function body will only be executed
when the button is pressed,
because the anonymous function is passed as a value
to the onPressed argument,
and that argument is used internally inside of that button
to then execute this function whenever the button is tapped.
And therefore this code in this function body
also only executes when the button is tapped.
Alternatively, we could also replace this anonymous function
with a pointer at startQuiz.
So this startQuiz variable, which we added here.
Keep in mind that in that variable we store a function,
and therefore, of course,
we can use this variable as a value for onPressed
because onPressed wants a function,
startQuiz contains such a function,
and therefore startQuiz can be used here
because it's then the value
stored in this startQuiz variable
that will be used as a value for onPressed.
And that is that function
that should then be executed when the button is pressed.
And ultimately it's this switchScreen function of course,

because that's the function we're passing around.

And with that, we in general have the setup we need to manage the state in that widget, that quiz widget, that is in the end a parent widget of both StartScreen and QuestionsScreen because both these widgets are used inside of the quiz widget.

But at the moment I'm getting an error.

47. The "initState" Method

Now, this error is related to me having const here, but even if that's removed, we still get an error.

We're getting this error here because I'm using switchScreen here when initializing this activeScreen variable.

And this basically happens at the same point of time where this method here is created, and therefore, it happens too early.

Both this variable initialization and this method creation here happens when this class is turned into an object.

So, you could say these lines here happen almost simultaneously, and therefore, here it's too early to reference this switchScreen method, because essentially it's not guaranteed that it exists yet, you could say.

So, how can we work around that?

Well, the best way of working around that is to add another method here to this QuizState class.

And that would be the built-in initState method.

If you start typing, you should get these auto-suggestions,
and if you complete this, this function here,
this method should be added.

Now, `initState` is a method that is, in the end,
provided by this `State` class, which we are extending,
but we can override it to add our own initialization logic.
And this indeed is a method that you typically wanna add
to your `State` classes.

So, to your classes that are based on `State`
to do some extra initialization work
when that `State` object is created for the first time.
`initState` will execute once
after this object has been created,
and therefore, in here,
you can indeed use all these variables,
and also reference this `switchScreen` function for example,
but `initState` will then never execute thereafter again,
which makes this the perfect place
for some general initialization work.

And therefore, it's actually here inside of `initState`
where I want to set
`activeScreen = StartScreen(switchScreen);` like this.

We don't need to wrap this with a `setState` here,
and we shouldn't wrap it with `setState`,
because this will execute
before the `build` method runs anyways.

So, therefore, `activeScreen` will be set to the right value
when the `build` method executes,
because `initState` will be executed first by Flutter.

Therefore, you don't need to call `setState` here
because we don't need to run the `build` method again
because it hasn't been executed

at all at this point of time.

And up here, we therefore now, set `activeScreen` to nothing,

and hence, you should add a question mark here

to make it clear that `activeScreen` can also be null.

Initially, it doesn't have a value,

because that will only be the case after `initState` ran,

which of course, is also pretty much the case

after the object was created,

but technically it's not the case

right from the very start the object was created.

Instead, `initState` is just called

immediately after the object was created,

but technically not during the object creation,

which is why technically `activeScreen` initially is null,

which is why you must add this question mark here.

And with that, we can now reference `switchScreen` here,

because in here, all variables and methods

will have been initialized and created.

You also should keep this `super.initState` call here,

because this makes sure that in the parent class,

in the `State` class, `initState` is executed as well

to allow Flutter to do its additional initialization work

that might be performed in that parent class.

And with that, if we now save this and force a reload,

we see this initial start screen,

which proves that setting `activeScreen`

to `StartScreen` here worked,

because it is `activeScreen`

that is then used down there as a child.

And if this would be null still, we wouldn't see anything,

so the fact that we see something here proves that it works.

And if we now click the Start screen,

we switch to the Questions screen.

This screen only has some dummy content right now,
which is why it looks so broken and ugly,
but it actually worked.

We switched to that screen,
and that proves that our logic here works,
that we successfully lifted up State.

48. Deep Dive: Flutter's (Stateful) Widget Lifecycle

Every Flutter Widget has a built-in lifecycle: A collection of methods that are automatically executed by Flutter (at certain points of time).

There are three extremely important (stateful) widget lifecycle methods you should be aware of:

`initState()`: Executed by Flutter when the `StatefulWidget`'s `State` object is initialized

`build()`: Executed by Flutter when the Widget is built for the first time AND after `setState()` was called

`dispose()`: Executed by Flutter right before the Widget will be deleted (e.g., because it was displayed conditionally)

You will encounter them all multiple times throughout the course - therefore you don't have to memorize them now and you will see them in action. It's still worth learning about them right now already.

49. Using Ternary Expressions & Comparison Operators

So we rendered content conditionally

and we successfully lifted our state up.

And of course for that,

we had to dive into some new patterns.

We had to add a new method here with `initState`.

But I hope it's clear how all these things work together

and why the app now works the way it does work.

Before we now continue and we, for example,

work on the actual questions screen,

I wanna show you an alternative way

of rendering our different screens here conditionally.

Currently we're making sure that different screens

are rendered by storing different screens

in this active screen variable.

This works and you can use this,

but this also requires us to add this initState method here,

for example, to set active screen to the proper widget.

We need it to use initState

because we need it to reference switch screen here

when storing that initial value in active screen.

And this was not possible when initializing it up here,

as you learned.

That's why it's also worth looking at another solution,

which will actually allow us to get rid of initState.

Instead of storing the active screen in a variable,

we could also store some identifier for a screen in here.

We could still call the variable active screen,

but now instead of storing the widget,

we could store a string.

For example, start screen as an identifier

for the start screen.

Though, this really is up to you.

It could be any string identifier.

It could even be a number.

Every screen could be mapped to a number.

But here I'll map screens to strings

since this is a bit more human readable, I'd say.

And then in switch screen,

we could therefore replace active screen,

not with a widget, but with another string,

which here could be questions-screen.

Again, this identifier's all up to you,

but both are strings now.

And what's the advantage of using strings?

The advantage is that here,

we can of course initialize active screen to a string,
and now we can infer the actual widgets
that should be rendered based on these string values.
And for that, we can go to the build method
where we need the actual widget that should be output.
And down here as a value for child,
we can now use a so-called ternary expression.
A ternary expression in Dart,
and in many other programming languages as well,
is a feature, a concept,
which allows you to check a condition
and then yield one value if the condition is met,
and an alternative value if it's not met.
And it's written like this.
You first define your condition.
For example, here,
that active screen is equal to start screen.
And that would be written like this.
activeScreen equals with two equal signs.
That's important. Start-screen.
Now it's two equal signs instead of one, because here,
we actually don't want to assign a new value
to active screen.
That is what you would do with one equal sign.
And here would be the wrong place for that,
because here a value is wanted, a widget value is wanted,
not some assignment operation.
And therefore we use two equal signs because two equal signs
are a totally different thing in Dart.
And again, in many other programming languages as well.
Because this is now a comparison operator
that checks for equality.

This operator will yield true as a value
for this overall code if active screen
is equal to start screen.
And otherwise it yields false.
And maybe you remember this slide
from the previous course section,
where I did mention these Boolean values already.
It's a special kind of value, which is supported by Dart
and basically all other programming languages out there
in the world, which is either true or false.
And this Bool type exists almost entirely
just for the single reason that often in your code,
you need to compare values,
and you must find out whether a certain condition
is met, true, or not met, false.
So here this code will yield true
if active screen currently holds,
start screen written like this as a string value,
and false otherwise.
And here to then turn this into a ternary expression
and use this information,
whether this is true or false, you then add a question mark.
And after this question mark,
you then define the value that should be yielded
by the overall ternary expression.
And that should therefore be used as a value
for a child if this is true.
And here that would then be the start screen widget,
which is created like this.
And to which we still pass a pointer
to the switch screen method.
So just as we used it before somewhere else,

now here in the build method,
derived conditionally based on this condition.
But at the moment in this condition,
we only have one value that we use if this condition is met.
We also must define the value that should be used
if it's not met.
And you do that by adding a colon thereafter.
And now you define the else case,
the value that should be used
if this condition yields false, if it's not met.
And that value here in this case,
would be my questionsScreen widget,
which can be turned into a const value.
Now I formatted the code and this is now
how the overall ternary expression looks like.
We have a condition with the double equals in operator,
which yields true or false,
and then we have a value that's yielded if this is true
and we have a value that's yielded
if this condition is false, so if it's not met.
And whichever value is yielded here,
so StartScreen or QuestionsScreen,
is then the value that's passed to this child argument.
With that, if you save that, and force your app to reload,
you should still see that start screen here initially.
And when you click this button,
it should still switch to the questions screen.
So this also works,
and that's a different approach
for rendering content conditionally.
It's up to you which approach you prefer,
but this approach has the advantage

that it requires a bit less code.

Most importantly,

it does not require this extra initState function.

And I'm showing these different approaches

because in different situations,

you might need different solutions.

And of course,

you also simply might prefer different solutions.

Now here I'll stick to this identifier solution,

but I wanna show you yet another alternative

to this ternary expression here.

50. Understanding "if" Statements

But now, I wanna make sure that screenWidth

is set to a different value

if activeScreen is not start-screen, but questions-screen.

To achieve this, we could cut this ternary expression

and use that up there.

And then we could actually even set this to final,

because now this never changes.

It's only set once whenever the build method runs

either to StartScreen or to QuestionsScreen,

and you could then use screenWidth

as a value down there for child,

so that the value stored in screenWidth

is passed as a value to the child argument

and that screen widget value is derived dynamically

with a ternary expression up here.

You could do that, but with that,

you would, of course, still be using a ternary expression,

though, now at least outside of the widget tree,

which already could be considered to be a bit more readable.

But I want to show you a different approach.

Hence, I'll go back to storing StartScreen

as an initial value in screenWidget,

and I'll set this back to be a variable,

because now I conditionally want to set screenWidget

to QuestionsScreen if a certain condition is met.

Now, for that, first of all,

the type here for a screenWidget

should explicitly be set to Widget

so that it's not inferred in a too-restrictive way.

We did that a couple of lectures ago already, if you recall.

But now this code here should be executed conditionally,

and for that, you can use another keyword provided by Dart,

the if keyword.

That's a crucial keyword, which you need to use a lot,

which is why I'm introducing it here,

because if allows you

to specify a condition between parentheses,

and then you add opening and closing curly braces,

and the code between these curly braces

will then only be executed if this condition is met.

So, now we could cut this code here

and paste this here into this if statement body,

and the condition that should be specified here

is now almost the same as before in the ternary expression.

We wanna compare activeScreen.

So, this variable,

which is dynamically updated if the button was pressed.

We wanna check if it's equal to

questions-screen now actually,

because we then set screenWidget

to QuestionsScreen if the identifier is equal

to questions-screen.

And here, we can add const.

And with that, whenever the build method executes, the screenWidth variable is initially set to StartScreen, but it then might be overwritten if this condition is met here.

So, if activeScreen is questions-screen, then screenWidth would be overwritten.

And that's an alternative pattern of setting content conditionally, because now screenWidth can of course, be used down there as a value for a child, and therefore, will now either be StartScreen or a QuestionsScreen that is set as a value for child.

And that's the last alternative I wanted to show you.

If we save this and force a reload, the app still works as before, but now using this approach.

And as mentioned before,

I simply showed all these different approaches, because we'll need them from time to time throughout the course.

And as a Flutter and Dart developer, you, of course, must know about these different approaches so that you can solve the different problems you might be facing when working on your own project, or also when working together with other colleagues that might be using one of these approaches.

51. Adding a Data Model & Dummy Data

So now that we learned about outputting conditional content,

it's time to continue with this app,
and here, with that questions screen,
which of course, is far from being finished here.
So for that, back in Code,
in this questions screen file here,
in the questions_screen.dart file,
where we have this questions screen widget,
the goal now in the end is to output a question,
and then a couple of possible answers for this question,
where only one answer will be right,
and after the user chose an answer,
that answer must be stored somewhere
so that it can be considered
for deriving the final results screen,
but immediately after picking an answer,
the next question in line should be shown with its answers
until there are no more questions and answers left.
That's the goal here,
so therefore, we need two main things here.
We need some widgets here to display the question
and the answers, but we also need the questions,
and I will start with that.
For that, in the lib folder,
I actually wanna add a new code file,
because I wanna define a class that acts as a blueprint
for my question objects,
because I wanna set up questions as objects
that group the question and the possible answers together
into one data structure,
because that's what objects are about in the end,
as you learned.
They are about grouping related data and logic together,

and even though we only used classes
to build custom widgets up to this point,
we can also use this class feature to create blueprints
for all other kinds of objects, like, for example,
these question data structures with which I wanna work.
That's why I'll add a new code file,
and I will actually add that file
in a subfolder in the lib folder.
So I'll add a new folder in this lib folder,
which you can, of course, also do by using this menu here
and then picking new folder,
and I'll name that folder models,
because I wanna store my data models in there,
the blueprints for the data I'm working with.
And in this models folder in the lib folder,
I'll add my quiz_question.dart file,
and this is now not a file that will contain a widget,
but instead, a file that will contain my question blueprint.
For that, I'll still use the class keyword,
because we of course, still need to create a class here,
and I'll name this class QuizQuestion,
though that name is up to you,
and this will now not extend anything,
because it will not be a widget, and there is nothing else
that this class would need to extend.
Instead, it's just defined like this,
and in this class, you can now define
which kind of data you wanna store
and which kind of additional logic you might want
to put in there.
And here, for a start, I'll add two final class variables,
instance variables, so two properties that should exist

on these quiz question objects later,
and the first one is the actual question text,
which should be of type string, defined like this,
and then as mentioned, I wanna have a couple
of possible answers, so a collection of answers,
and for that, I'll add a answers property,
but what could now be the type of that?
Well, in the end, it should be a list of answers,
a list of strings,
and therefore, the proper type here is list,
and then a list of strings,
utilizing this generic list type,
which I did show you a couple of times before
in this course already.
So that's my answers property,
and now, in order to set these properties to some values,
we also need a constructor function
so that we can reuse this class
and create different quiz question objects
with different texts and different answers
based on the same blueprint,
therefore, here, just as for our widgets,
we add a constructor function
that can receive some arguments,
which are then mapped to these properties,
and for that, I expect to get my text,
which is then mapped to this text property
by using this text,
and as a second argument here in this constructor function,
I expect to get my answers list.
And here, we can now add const, because indeed,
once such a object is created, in the future,

it will not be changed thereafter.

At least right now, there is no way of changing this,

hence, we can also allow the creation

of const quiz questions,

and we do allow this by adding const

in front of the constructor function.

With that, we got this quiz question class,

and now we can add another file,

which I'll add in another subfolder, in the lib folder,

which I'll name data,

where I wanna store my actual dummy questions

which I wanna use for this quiz here,

and hence, I'll name this file questions.dart.

The name is up to you, but in here,

I now wanna define all these questions,

and the idea here is to create a questions variable

or even a constant, since I don't plan

on changing that list of questions

as part of this application,

which could be named questions,

and that should then be a list of such question objects

that are created with help of this quiz questions class

on which we just worked.

For that, in this questions.dart file,

we have to import this model file on which we just worked

from our project, and they are from models,

this quiz_question.dart file,

and then here, in this questions list,

we can create our quiz questions

by simply calling the constructor function

and then passing the question text here as a first argument,

for example,

"What are the main building blocks of Flutter UIs?"

Like this, that could be my question,
and then as a second argument to quiz question,
we pass a list of answers, a list of answer strings,
because quiz question is defined such that we want
to get a list of strings as a second argument,
and that list of strings will then be mapped
to this answers property, as defined in our class.

And here, we could then have widgets
as the first possible answer,
components as the second possible answer,
blocks as the third possible answer,
and functions as the fourth possible answer,
and my goal here is to make sure
that it's always the first answer that is correct,
because we'll need to know which answer was correct
for showing an appropriate results screen later,
where we inform the user
which answer would've been correct
and which answer was chosen by the user,
and to make sure that we later know
which answer was correct.

I'll make sure that it's always the first answer
in this list of answers that is the correct one,
and to make sure that our quiz doesn't become predictable,
we'll shuffle these answers
before displaying them on the screen, but internally,
the correct answer will always be the first one
in this list of answers.

Well, and now you find a questions.dart file attached,
which you should use to replace your file with,
because this file, which you find attached,

then contains a bunch of example questions,
a bunch of dummy questions,
which we can use for this demo app here,
and I'm providing this file so that you don't have
to come up with all these questions and answers
and write all these questions and answers on your own.
The attached file assumes that we have a quiz question model
as defined in this `quiz_question.dart` file,
so in order to use these attached questions,
you must make sure that your quiz question model
looks exactly like mine,
so it must have the same name and the same properties.
And with that, we got the questions for our app.
Now, it's time to use them on this questions screen.

52. Configuring a Column

So, in this `QuestionsScreen`,
I now wanna output these example questions,
one after each other.
And for that down here in this `_QuestionsScreenState` object,
which is where we have this build method,
we of course, wanna output the question text,
and then below that, the answers to the question.
So, we have a couple of widgets above each other here,
and that of course, screams for a `Column`.
And this `Column` then has a couple of child widgets
passed to it through that `children` prop here.
And the first child at the top should be a `Text` widget,
which is used to output the actual question text.
So, here with this widget, we'll output, The question,
and we'll use the actual question text soon.
Below that question text,

I then wanna have some spacing and then a couple of buttons where each button represents one selectable answer.

So, to add the spacing,

we can use the good, old `SizeBox` again,

and set it to a height of, let's say, 30 maybe.

And to get rid of these blue squiggly lines,

we can add `const` in front of this list

to make sure that all the items in the list

are turned into `const` items,

and therefore, they are only stored in memory once

and can be reused by Dart, as you learned.

And below this `SizeBox`,

we then need our buttons for the different answers.

Now, here, I wanna use the `ElevatedButton` widget,

which is another widget provided by Flutter.

And for the moment, I'll just set `onPressed`

to an empty, anonymous function.

We'll make sure that we do something

more useful here later.

And now we actually have to remove `const` here

because the button isn't `const`,

and we have to add it manually

in front of the widgets that can be `const`,

`Text` and `SizeBox`.

And for the child here for this `ElevatedButton`,

we wanna use the `Text` widget

to output some text as part of the button.

So, by combining these two widgets as we did it before

for the `Text` button

and the outlined buttons as well in this course.

And here, the `Text` would be answer 1 for the first button.

And of course, this can now be repeated

so that we also have a second answer,

and a third answer and so on.

Now, soon, this will be generated dynamically

to really use the actual answers

of the question we selected.

For the moment, I'll just hard-code it like this.

If we now save this and force a reload

and we go to this Questions page,

it looks like this.

And of course, that's not the final look I wanna have.

Instead, I wanna make sure

that we take the full available width here for the column.

And, I again, wanna make sure that my question

and the answers are centered.

Now, for that, we could use the same approach

as we do on the start_screen

where we wrap column with a Center widget

and then set the mainAxisAlignment on Column

to MainAxisAlignment.min.

This would work here as well,

but to show you also an alternative,

we can wrap this column widget with another widget,

and that would be the SizedBox widget.

And here, I now wanna set the width to a special value

that will take as much width as is available.

So, instead of hard-coding this to a width of 100 pixels

or anything like that,

which will probably not be the exact width

of the device we're running on,

I can use a special value built into Dart,

and that's double.infinity.

Now, that's a special value which simply means,

use as much width as you can, be as wide as possible.

It's built into Dart,

and we can use it as a value for width here.

And with that, you will see that we now

at least take the entire width.

To center the questions and the answers vertically,

we then can use another setting inside of column,

and there, we can set the `MainAxisAlignment`.

You might remember that for `mainAxisSize`,

I mentioned that the Main Axis for Column

is the Vertical Axis from top to bottom.

And hence, the `mainAxisAlignment`

also controls the vertical alignment,

so whether it's centered

or if it's at the top or the end of the vertical axis.

And we can set this to one of the built-in values,

for example, `MainAxisAlignment.center`.

And if we do this and save that,

we now see that this is centered.

So, this is an alternative

to using a center widget as a wrapper

and setting the main axis size.

And one possible advantage of this approach

is that you can set the `mainAxisAlignment`

not just to center, but for example,

also to `spaceAround` or `spaceBetween`.

With `spaceAround`, for example,

the items would be distributed equally across the screen,

including the size widget,

which is invisible, but also takes up its space.

It's not what we need here,

but it could be another reason

for using `mainAxisAlignment` in certain situations.

Here, it's center, which I want though,

so that it looks like this.

Of course, the colors and so on are not final yet though.

And that, therefore, is the next step.

I wanna change the question color

and of course,

also the look and styling of my Answer buttons.

53. Creating a Reusable, Custom Styled Button

Now for the question color,

we can simply add the style argument here to the text widget

and set this to a value of type `TextStyle`

with help of the text style constructor function,

and set this to a color of white, like this.

With that, we got a white text color.

So that wasn't too special.

But for the buttons here,

I now wanna make sure that I can style them such

that I don't have to copy and paste

the same style to all my buttons,

because at the moment here, I have three buttons,

and if I wanna apply some style here,

for example, to the first button,

I can, of course, do that with the style argument,

and then by using elevated button style from,

as we also did it for the outlined button

on the start screen, for example,

but I would have to repeat that for all my buttons here.

And that quickly gets annoying,

and if I ever want to change the styling,

I also have to do that for all buttons.

And that's why this here is a great example
and a great use case for creating another custom widget
so that we have another reusable building block
that has this styling that we want for our buttons built in.
And for that, I'll add another file here in my lib folder,
and I'll name it answer_button.dart, like this.
And in there, we can create a new class, AnswerButton,
and here, since I want to build a widget,
I will extend one of the widget classes,
and for this button, I can extend the StatelessWidget class
because even though it's a button that can be tapped,
I have no state that would need to be managed
inside of that button.
Therefore it's a stateless widget, and of course, as always,
we import flutter/material.dart here,
and we add this build method, as we always do,
accepting this context argument
with or without this type annotation, doesn't matter.
And then inside of the body of this build method,
we return our widget, or a combination of widgets,
and in my case,
it's this elevated button which I wanna return there.
So I cut this from questions screen,
and instead return this here with a semi colon at the end,
and now we'll be able to set up our styling here,
and then reuse this answer button in the questions screen.
So in this column here,
we'll be able to use the answer button a couple of times,
and all these buttons will then have the same styling
because the styling is locked into our custom widget.
That's the plan.
Now of course, the text that's set on this button

should be set from outside this button though.

It should not be locked into this button.

And of course, you might already know how this works because you learned how to make your widgets configurable in the previous core section as well.

We can add a constructor function here, answer button, and a property, in this case, the answer text for example, which should be a string, and then here in this constructor function, we can accept and set the answer text argument, and also accept and forward that key argument here, this named key argument, and then we can use answer text, which we get as an argument and which we store in this class variable down there as a value for this text so that this text is actually set from outside this widget.

This now can't be const anymore though because answer text is now a variable, actually one that doesn't change, but still, the text widget can no longer be used in conjunction with the const keyword here.

That's not all I want to accept here though.

Instead this function, which is set as a value for on pressed on the button, should also be coming from outside.

So it should also be settable from inside questions screen so that we can then in the end pass our answer text to this answer widget, and then as a second argument, also our function that should be executed when the button is pressed, for example.

And to achieve this, we must accept another argument, a positional or named argument, that is up to you,

as part of this answer button.

And for that I'll add another property here,

which should be of type function

that actually returns no value.

So I want to get a function that returns no value,

and I'll name this one here onTap.

The name is up to you though.

And accept another argument, which is also named onTap,

and which is then automatically used

as a value for this property

by adding this dot in front of it.

And then it's this onTap variable

that can be used as a value for on pressed here.

On tapped is the function which we receive as an argument,

and which we then use as a value

for on pressed on the elevated button, and therefore,

we'll be able to pass a function here to the answer widget

when we use it in another widget,

like here in questions screen,

so that we can define which function

should in the end be executed when this button is pressed

from inside the widget

where we use our custom answer widget.

And of course, all these positional arguments here

could also be set up as named arguments.

And just to also show you how this would work,

I'll actually convert these positional two named arguments,

not because you have to,

but because it can be interesting

to see how this could be done.

You simply place these arguments between these curly braces

in your argument list here then,

and now you must add required,
the extra required keyword in front of them,
because by default in Dart, named arguments are optional,
but here we need values
to make sure that this answer button widget works correctly,
and therefore, we must make them non-optional
by adding the required keyword
in front of the named arguments.

Besides that, they now work like positional arguments.

They can receive values

and these values are then forwarded

thanks to the this keyword

to the properties that have the same names.

So answer text to answer text and onTap to onTap.

Now we can turn this into a const constructor function.

This is possible here now for this widget.

And therefore now in questions screen,

we can use this widget, by first of all adding an import

where we import from advanced basics,

and then from answer button dot dart.

And with this import added down here,

you can now use this answer button widget.

And since I'm now expecting named arguments here,

you have to target these arguments by name.

So answer text and onTap.

And you could also change the order here now

because for named arguments, this order is not important,

it's just important that you set all required arguments

and that you target those arguments by their name.

That being said, I'll go back to the other order

because that makes a bit more sense for me here.

And with that, we now could reuse this answer button here

for answer one, and answer two, and answer three.

But of course, we haven't changed the styling yet,
so that's what I'll do next.

Now, to change the styling,
I'll go back to my answer button widget here, and there,
I already added the style argument to elevated button,
and I'm using the style from method on elevated button
to easily generate such a style object.

But now we need to pass some arguments to style from
to really change the styling.

And for me,
that changes with the background color of these buttons.

Here I want to use a color,
maybe some purple color,
but actually I'll use that color picker again
to pick a dark bluish purple color here, maybe,
something like this.

And we can now add const in front of that.

And I also wanna change the foreground color,
so the text color,
and set this simply to white, like this.

If we now save that, all buttons have this style,
even though we changed it just in one place,
because we did it in our custom widget,
which we're reusing a couple of times.

Now that's a good start,
but I actually also want to change the shape
of these buttons a little bit.

And for that, we can set the shape argument here,
still in this style from method,
and set this to a specific shape
that is actually also suggested here.

If I open this suggestion menu by pressing Ctrl space,
and here it's this rounded rectangle border shape,
which I want.

With this constructor function,
I can create a configuration object for my buttons
that allows me to add some nicely rounded corners
to the button.

For that, this rounded rectangle border constructor function
wants a border radius argument,
which can be set to border radius circular
using another class and constructor function
provided by Flutter.

All these things here are provided by Flutter,
and border radius circular simply adds some rounded corners,
and you can set the radius,
and you can simply try out different numbers here,
but 40 should achieve the look
which I want to have here for this application.

With that, if we save this,
we have some nice rounded off corners,
and the buttons should now look something like this.

And now I just want to add more padding,
so more internal spacing
between the text and the borders of that button.

So that padding here should be increased.
and that can also be done with this style object in there,
anywhere in this style from method.

You can add the padding argument
and set this to a value with the Edge Insets class,
provided by Flutter, and you could use `EdgeInsets.all`
to apply the same padding in all directions,
but here I actually want to have

different horizontal and vertical padding,
and this can be achieved
by using the special symmetric constructor function.
This constructor function
allows you to set separate paddings
for horizontal and vertical padding,
and here for vertical padding, so top and bottom,
I want to have a value of 10,
and set horizontal to 40 maybe.
And now we can add const in front of this.
And if you now save this, the button should look like that.
And of course you can also change them,
tweak this a bit more to your liking,
but of course, keep in mind
that this is all just some dummy text right now,
which will soon be replaced with the actual question text
and the actual answers.
But for me right now, that is the look I want to have,
and I'll tweak it a bit more later, but as a next step,
I'd actually like to use the actual question text
and the actual answers that we prepared
in this questions Dart file.

54. Accessing List Elements & Object Properties

So now the goal is
to output the actual question data,
the actual question text,
and its answers on this screen.
Now, for that,
we, of course, got our questions list here
in this questions dart file.
But how do we now output the texts and the answers

for these different questions

in this questions screen widget now?

Well, we can start

by importing the questions into this file.

That's a good start because that at least allows us

to use this list in this file.

So for this, we can add a import here

where we import from our own project,

and they are from the data folder

and there from the question start file.

And with this import added,

we can access this questions list here.

Now we just need to access the different elements

in that questions list.

And therefore, it's important to understand

how you can access list values in Dart.

And for that, let's take a simple example,

a list of hobbies, a list of strings, therefore.

And let's say in some place of your code,

you want to access the first element in this list.

So, cooking in this case.

This could be done by using this notation.

This special notation here

allows you to use a list index starting at zero

to access the different list items.

So since this list index starts at zero,

you would use this notation to access the first list item

and this notation to access the second list item.

So the first item in a list has an index of zero,

the second item has an index of one and so on.

And this index can be used

with this special square bracket notation

after the name of the list.

So after the name of the variable

or constant that contains the list to be precise

to access the different list values.

And if you would try to use an index that doesn't exist,

so in this case, for example, the index two

for a list that only has two elements,

you would get an error.

So you can't access elements that aren't there,

which hopefully makes sense.

Now, for us, this means

that we can, for example, access the first question

in this list of questions

by using this questions constant,

and then access the element with an index of zero.

So in our questions screen widget here in the build method,

we can get our current question, for example,

which is stored in a final variable here

since I don't plan on reassigning it

inside of this build method.

By accessing questions, which is this questions list here.

So it's named questions here in this file,

which we are importing.

Hence I can reference this list by its name here

in my code in that file where I have that import.

And the first question then would be accessed like this

with square brackets zero.

That accesses the element with an index of zero

in that list of questions.

And now we could use this current question here

to output the text instead of placeholder text.

For that, we can use current question,

this variable which holds this question.

And now it's important to keep in mind

that our questions are objects

created with help of this class constructor function.

And in dart, you can access data stored in objects

by using the dot notation,

which you also already saw from time to time in this course.

By adding a dot after a variable

or constant that contains an object,

you can access the available data properties

and methods that exist on this object.

In this case, I could access the answers,

but I can also access the text of this question.

Now, this can't be const anymore here,

we have to remove const

because this is now derived dynamically,

but at least the text style can still be set to const.

And we can now of course also set

our answers dynamically here.

For that, for the first answer button,

we can access current question.answers,

and now answers is yet another list.

So of course, here we can again use

this square bracket notation to access the first answer.

And then for answer two,

to access the second answer by using the index one.

And that can be repeated, of course.

We can do this for the third answer here,

and then add a fourth answer

by accessing the answer with index three.

And if you do that and save everything,

you will see that you got your question here

and then your different answers.

And that is indeed how you can access this question data.

Though that's not entirely how I want to use that data.

Because for example, with this solution,

I manually created all these answer buttons,

which is a lot of unnecessary code duplication.

And as developers,

we typically try to avoid code duplication

because that then makes it easier for us

to change things in the future.

And it also has the disadvantage

that if we had a question that had more than four answers,

which is not the case here,

but which we could of course have,

then we would not have enough answer buttons here,

or we would have some empty answer buttons

for questions that have less than four answers.

So hard coding these answer buttons here is not ideal,

and that's why we will output these buttons differently

in the next lecture.

55. Mapping Lists & Using the Spread Operator

The goal now is to generate

these AnswerButtons here dynamically

based on the actual amount of answers

that we have for the current question.

And by the way, later we will of course also make sure

that the question changes once it has been answered.

But for the moment,

we just wanna output these AnswerButtons dynamically.

And this can be done by going to our children list here

in this column,

and then in there we can access

`currentQuestion.answers`, which is this list of strings.

And now we must convert these strings to widgets, so to say,

to widgets that then use these strings.

And such a conversion can be done

by using Dart's built-in `map` method.

Now for that,

it's important to understand that `answers` is a list,

it's a list of strings.

And lists no matter which kind of values they contain,

but lists in general in Dart have a broad variety

of built-in methods,

built-in functions that can be called on them.

And these functions simply interact

with the list in different ways.

For example,

`add` would allow you to add a new item to that list.

Something we'll also use later in the course.

`Map` allows you to convert, to transform you could say,

the values in a list to other values.

For that `map` takes a function as a value,

so that's another example for a place

where a function is used as a value,

and this function which you pass to `map`

will be executed automatically by Dart

for every list item.

So Dart automatically goes through the entire list

and executes this function on every list item.

Now Dart will then also automatically pass the list item

for which it is currently executing this function here

as argument to this function.

So here you get the list item

for which this function is currently being executed,
and inside of the function body
you then can return the transformed item.
So the new value you wanna have in the list
instead of that old item.
And of course, that new value which you return here
can and typically will be based on this item
on which the function was called
and which you receive as an argument here,
because that then allows you
to basically replace all the items in a list with new items
that are based on the old items.
And here I wanna return my `AnswerButton` widget
so that I convert this list of answers,
this list of strings to a list of `AnswerButtons`.
And since widgets are in the end just objects
which are totally normal values in Dart,
it's absolutely fine to convert a list of strings
to a list of widgets,
even though it might sound strange at first.
Now, it's important to understand
that this conversion does not change the original list.
So the list of answers that is stored
in these quiz question objects
does not change when calling `map` here.
Instead, `map` in the end yields a new list,
a brand new list object in memory,
but both lists coexist in memory.
That's really important to understand.
`Map` does not change the original list,
instead it returns a new list full of new objects.
In this case, full of new `AnswerButtons`.

And on these AnswerButtons,
we can of course then set answerText to item,
because item here will be of type string,
since our answers are strings in that list of strings.
So item is our string.
We could also name it answer instead of item
because that name is up to you to make it clearer
that we'll get the individual answers here
since this function is executed
for all the answers automatically by Dart,
and hence, we can set this answer string as a value
for the answerText which wants a string.
And for onTap, we can for the moment,
just pass an empty anonymous function.
But now you see I'm getting an error here,
and in the end I'm getting an error here that this element,
which is of type, iterable, whatever that may be,
can't be assigned to the list type widget.
The problem which we in the end have here
is that I now got a list of AnswerButtons
or something like a list,
it's actually an iterable as it turns out.
But you can treat this almost like a list here.
But Flutter isn't happy with getting a list
or an iterable here.
And that would also be the case
if I manually create a list here,
a list of AnswerButtons for example.
Here, I also would be getting an error
that this type of data is not wanted here.
And it's not wanted here because children, this argument,
which we set on the column here, wants a list of widget.

Now here inside of that list that we are creating,
we are creating another list though, a nested list.
So now children is a list of widgets
like text and size box,
but also a list that contains other lists as values,
and that's simply not what children wants.
It wants just widgets as values inside this list,
not other lists.
So converting our answers to AnswerButtons is not enough
if there is still in a list thereafter.
Instead, what we have to do here is something
called spreading.
We have to spread this list or this iterable,
which is generated by map,
into a couple of individual values.
And that's done by adding three dots in front of this,
which might look strange
but which is another built-in operator,
a built-in keyword you could say which is available in Dart.
And these three dots will simply take all the values
in a list or an iterable,
and pull them out of the list,
pull them out of the iterable,
and instead place them as comma separated values
in this place in code here.
So now these AnswerButtons in this list
of AnswerButtons here are pulled out
and added as individual comma separated AnswerButtons
to this list here which is set as a value for children.
That is what's happening here with this code.
And with that,
we can and should now get rid

of these extra AnswerButtons here,
and just use these AnswerButtons here
which are created based on the answers list,
which is a list of strings,
which is then converted with help of map
from a list of strings to a list of AnswerButtons,
and then these AnswerButtons are pulled out
and added as individual elements to this children list.
So that overall here we get a list of widgets.
And with that, if you save everything,
you see that, again, here,
we have all these answers like before,
but now they're dynamically generated
instead of being hard coded.
And this is a super important pattern
because being able to dynamically generate
an output list content is crucial
since you will see that most apps
deal with some kind of list data in some parts.

56. Alignment, Margin & Padding

So now that we're deriving
these answer buttons dynamically
based on the actual answers in our list of answers here,
there are three things I want to change
regarding these answers.
For one, the styling.
I'd like to make sure that these answer buttons
take up all the available space,
leaving some empty space to the left and right.
That would be nice.
In addition, I wanna shuffle them

because, as mentioned earlier,
it's the first answer that is the correct one.
And knowing this will be important
for later presenting the results.
But for showing the answers,
I actually wanna shuffle them
so that it's not always the first one that's correct.
And third, I, of course, wanna make sure
that when we tap those answers,
we proceed to the next question
or, after going through all questions,
to the results screen.
Now let's start with laying these things out correctly.
To make sure that these answers
take all the available space,
we can go to the column widget here.
And besides setting the `mainAxisAlignment`,
we can also set the `crossAxisAlignment`.
The main axis in a column is the vertical axis.
The cross axis is the horizontal axis, therefore.
And if you set this to `crossAxisAlignment.stretch`,
which is a built-in alignment option for the cross axis,
the items inside of the column
will actually stretch out to be as wide as possible
to fill the entire column width.
But of course, that now also means,
as you can see, that there is absolutely no space left
to the left and right.
And that's also not what I want here.
To make sure that there is some space left,
we, therefore, have to add some padding
or margin around the column.

And here it won't really make a difference

which of the two things we use here.

In theory, a padding is the internal spacing

between content and the borders,

as we have it here for the button.

That would be the padding.

And the spacing between widgets,

so this spacing here, for example,

would be the margin around the widget.

Now visually, it here doesn't matter

what we set for the entire column

to add some spacing around it.

And therefore here, I'll go for a margin

by picking this column here

and refactoring this to wrap it with another widget,

with the container widget,

because that container widget

then has a margin argument that can be set.

And margin like padding,

which you already saw earlier,

takes this `EdgeInsets` class here provided by Flutter.

And with `EdgeInsets.all`,

we can add the same margin in all directions.

So top, bottom, left, right.

And here we could set this to 20.

That can also be turned

into a constant value, thereafter.

So with that, we now have some spacing.

Maybe I'll actually use 40 here.

Yeah, looks nicer.

And I also wanna center this text here.

And to do that, we can go to this text,

this Text widget.

And there on the text,
not on the text style, but the text itself,
you can set a TextAlign argument,
which then can, for example, be set
to TextAlign.center to, as you might guess,
center the text in the middle horizontally
so that now it is centered.

So that's now the visual part.

And I'll get back to this text here,
for example, later again.

But for now, I'm quite happy with how this looks.
But of course, the answers are still not shuffled,
and that's, therefore, what we'll tackle next.

57. Mutating Values in Memory

So how can we shuffle these list items here?

Well, that thankfully is easy enough to do
because Flutter gives you a shuffle method
that can be called on lists.

So just as map could be called on lists,
shuffle can also be called on lists.

It's a built-in method that can be called on lists
to change the order of the items in the list.

However, what's important to understand regarding shuffle
is that unlike map, it does change the original list.

Map does not change the original list.

Shuffle does.

And I actually don't wanna change this list of answers.

I wanna make sure that the correct answer
stays the first answer here,
and therefore we have to first copy this list

before we shuffle it.

Otherwise, finding out which answer was the correct answer will impossible after these answers have been shuffled.

So therefore, here, to make sure

that we're shuffling the answers in a brand new list,

we can go to this QuizQuestion class,

which we added earlier,

which is our blueprint for these questions.

And in there we can add a function, a method

that could be called getShuffledAnswers like this.

This should return a list of string values,

so the same data type as we used it here for answers.

And inside of getShuffledAnswers,

I therefore what I call this shuffle method on answers.

But I first want to create a copy of answers.

And creating such a copy is thankfully straightforward.

We can use the built-in List class,

so List is not just a type but also a class built into Dart.

And this offers a couple of utility constructor functions

that make it easy to create different lists.

For example, it has a of constructor function,

and this of constructor function creates a new list

based on another list.

So it copies an existing list basically.

And therefore here we can pass answers to List.of

to create a new list based on that old list of answers.

And then we can call shuffle on this new list.

So now the new list will be the shuffled one,

and the old list will be untouched

because we copied the list and then shuffle the copied list.

And what we're doing here, by the way, is called chaining.

We're calling a method on the result

of calling another function or method.

And this is something you'll see quite a lot when working with Dart and Flutter.

It is quite common

to chain command after command after command.

Now here it's of course not enough

to just shuffle the list though,

instead we also must return it.

And here it turns out that shuffle indeed returns nothing.

So shuffle does not yield the shuffled list.

Instead it shuffles the list on which you call it in place

and does not give you the updated list.

Therefore, here, to get hold of that updated list,

we actually have to change the code a little bit

and first create a new final variable,

which could be called shuffledList,

which is initially just the copy of the list,

but then we call shuffle on that shuffledList.

And then it's this shuffledList here, which is returned.

And we must do it like this

so that we first create this copy

and store this copy in a variable,

and then we call shuffle, which as mentioned,

changes that list in place.

So after this line of code, shuffledList will have changed,

and then we return the updated shuffledList.

And you might wonder how a shuffledList

can change if it's final,

but final only ensures that you can't reassign the variable.

So I can't, for example,

assign shuffledList to a new list here.

This does not work because of final,

but that's not what we're doing when calling shuffle here.

Instead, this reaches out

to the existing list object in memory

and edits the items in that list.

It does not override the list as a whole.

So calling shuffle here does not violate final,

because we're not reassigning a value to shuffledList,

we're just editing an existing value in memory with shuffle.

And with that,

we now got our getShuffledAnswers method here

and back in questions_screen,

we can now use these shuffled answers.

We can use them down here.

Instead of accessing currentQuestion.answers,

we can access currentQuestion.getShuffledAnswers(),

and then map those shuffled answers to our answer buttons.

So this logic doesn't change.

We still chain this map method here

onto the getShuffledAnswers method,

but the difference now is that we map our shuffled answers

and very important we do all these things

without changing the original answers list.

With that, you'll see that now I have a different order

than I did before.

Before, Widgets was at the top

because this is the right answer.

Now here, for me, its Functions,

and the next time I'll visit this page,

it will likely look different again.

58. Managing The Questions Index As State

Now with those shuffled answers

being displayed here,
it's time to make sure that once we select an answer,
we move on to the next answer in line.
And to do that, we must make sure
that this question index here changes over time
because this question index here determines
which question is our current question,
which then determines which question text
and which question answers are shown on the screen.
So how can we change this index?
Well by managing it as state again,
and that's why I created questions screen
as a stateful widget
so that we can manage and change state in there.
We should now add a new variable here
in this questions screen state class.
And that could be the current question index.
The name is up to you,
but of course it should describe which kind of value
will be stored inside of it.
Now initially this should probably be set to zero
so that we start with
the first question in our list of questions,
but it should then change
whenever we click one of these answer buttons.
Therefore, we also must add a method
to this questions screen state class here,
and that method could be called answer question
because that is what we're in the end doing
when we click one of these answer buttons,
we answer the question.
And as a result of doing that,

we wanna move on to the next question.

But of course later we also wanna make sure
that we store the answer that was picked.

So therefore here we have the answer question function,
which doesn't return anything.

And in there I want

to increase current question index by one.

Now this can be done by setting current question index
equal to current question index plus one.

This might look strange at first,

but you can access the variable,

which you then set on the left side of the equal sign,

also on the right side of the equal sign

to derive the new value to which it should be set,

because that right side will actually be executed first.

So here, when this line of code executes,

Dart will first take a look at current question index

here on the right side of the equal sign

and take the value that's currently stored in there.

Initially that will be zero.

It'll then add one to it so that overall

for the first time this function is executed,

we would have a value of one here.

And then this new value is stored back into this variable.

So that is how we can increment this variable.

And since this is something you will find yourself doing
quite often in code,

Dart actually also gives you a shortcut for this.

Instead of writing it like this, which you can do,

but which is a bit long,

you can also set current question index plus equal one.

This code is equal to this code.

By using this plus equal operator,
You are telling Dart that you wanna set this variable
equal to its current value plus one.
So this is simply a shortcut for this.
And if you want to increment the value by one,
there's an even shorter syntax
because you can then also just set
current question index plus plus.
This increments the value by one, and only by one.
So if you wanna add two to that value,
this syntax could not be used.
This syntax could be used though.
But if it's one that you want to add,
you can use this syntax, this code for doing that.
And a similar syntax would also exist for deducting one
by using minus minus, or here in this case, minus equals.
And that's not just a case in Dart.
Instead, you find these syntaxes
in most programming languages.
So this will add one to the current question index.
And now we can use the current question index here
instead of hard coding zero inside of the build method
to access the question with that index here.
Now, of course, at the moment,
the build method would never execute again though,
because as you learned before,
you must call set state,
the set state function provided by flutter,
in order to tell flutter
that the build method should be executed again.
So that's the correct way
of updating current question index.

And with that, the next step is to make sure that this answer question method here is actually executed when our button here is clicked.

Now, to achieve this, we just have to replace this empty anonymous function, which I used as a dummy value for my onTap argument here, which wants a function, with a pointer at this brand new answer question function, which we just added.

As you learned, without parentheses, because we wanna point at that function and pass the function as a value to onTap.

And with this, if we now are saved this all, we should be able to click this and move on, move between those answers.

So that's working as it should.

Of course, at some point we'll reach an error here because at some point we try to move on to a question that doesn't exist because we never stop incrementing this current question index.

But we will of course take care of that soon as well.

59. More on Button Styling

So at this point in the section, we can start the quiz and we can move between our questions here by answering them.

And you might notice that some answers look quite ugly because the text is not centered.

Basically all answers that span multiple lines are not centered,

and that is something I want to change.

And I also wanna change

how this question text here looks like,

because I think it could stand out a bit more.

But let's start with centering this text.

To center this, we can go to our answer button widget,

and there, we have this elevated button

which has this child widget here,

which is the text widget that is responsible

for displaying the text in this button.

And of course here we can add text align as we did it

for the question text a couple of lectures ago.

And we can set text align to text align.center here

to make sure that all text is centered.

So that's one thing.

Now, regarding this question text here,

I wanna do two things.

I want to make it a bit bigger and bold,

but I also want to use a different font family.

60. Using Third-Party Packages & Adding Google Fonts

So my next goal

is to adjust the styling of this text here,

the question text,

and as mentioned,

I want to use a different font family

for this text as well.

Now, if you don't want to use the default font here,

as we're currently doing it,

there are different ways of adding custom fonts.

And attached, you'll find a link,

which would explain you how you could add custom fonts

by importing font files manually into your project folder,
how you could then register these font files
in your pubspec folder,
and then use them in code.

You can do that,
but one of the easiest ways of adding custom fonts
to Flutter projects is by using Google Fonts
and the special Flutter google_fonts package,
which you can add to your project.

Now, google_fonts is, of course,
a very popular collection of fonts
that you can use in all kinds of projects,
and adding them to Flutter is super easy
because of this official google_fonts package,
which you can add to your project.

Now attached, you will also find
a link to this package here,
where you can learn more about the package
and how to get started with it.

But of course, in this lecture,
we're also going to add and use it together.

Now, adding extra packages to Flutter projects
is something you'll typically do quite a lot
when building Flutter apps
because there are different packages
for all kinds of things.

And you will see various extra third-party packages
being added to Flutter projects
throughout this course.

This pub.dev site is the official Dart package site,
which is used for hosting
and listing all those third-party packages

which you could add.

And for every package,
you find an entry that looks something like this,
where you got a Readme site
with various pieces of information about this package,
where you find a Changelog telling you what changed
with the latest versions of that package
that have been released,
where you might find some examples,
and where you find some installation instructions.

And that's now the important part for us here
because as a first step,
we'll have to install this package.

And this can be done by copying this command here,
which you see here.

With that copied,
in your project,
you should open your terminal,
and in Visual Studio Code,
you can open this terminal
by going to View, Appearance, Panel.

And this is then your default System Terminal
just integrated into Visual Studio Code
and already navigated into this project folder.

And there you can now run this command,
which you just copied from this package site.

In this case, `flutter pub add google_fonts`.

Now this command will now install this package
into this project,

and as part of this installation process,
once it's done running that installation process,
it will also add the `pubspec.yaml` file.

In that file,
you should now find this new package added here
under dependencies.
So here's the dependencies
and nested below the dependencies,
you will find this new package with its latest version,
which, of course, might be a different version
at the point of time you are viewing this.
Now with this added,
we can use it,
and you'll learn how to use it here
in this Readme part of the package page.
But, thankfully, using it is also super simple.
In the end, you go to the Text widget
where you wanna use one of these Google Fonts.
So here, for me, for example,
that's in the questions screen widget.
And there it's this Text widget here.
And you replace the existing style object here
with a new object
that is imported from this google_fonts package.
For this, of course,
you must add a import at the top,
where you import from this google_fonts package
and there from the google_fonts.dart.
This must be added
in order to use these google_fonts package features
in this file.
And with that import added,
down here where I just removed this style object,
you can now use GoogleFonts.
and then you have a broad variety of methods here,

a broad variety of constructor functions, to be precise,
where these different functions map
to different font families.

And you can visit fonts.google.com
to get an overview of all these available font families
and to find their names.

And for example, for me here,
I want to use this Lato font here.

And in order to do that,
I do for axis `GoogleFonts.lato`
and execute this like a function.

And this then allows you to again,
set all these values you could have set
on that text style object before as well,
so for example, to apply a text color.

But in addition to those settings,
it will now also apply this font family,
so the Lato font family, in my case here.

And here, in addition to setting this text to be white,

I also want to change the font size
and set this to maybe 24,
and change the font weight here
and set this to `fontWeight.bold`,
which is one of the predefined font weight values
you can use

because the font weight argument wants a font weight value.

With that, if we save this,
you now see that the question text looks differently,
that a different font is used,
and that it's a bit bigger.

Now, I also wanna tweak the color a little bit,
and for that, I'll hover over `Colors.white`,

and I'll pick some light purplish color,
maybe like this.

Yeah, I think that looks a bit nicer now.

I'll just also turn this into a const value here.

And with that, you now also know
how you can add third-party packages
and how you can change the font families.

Of course, you can now not just do this here
for this question text,
but we could also do it on the start screen.

For example, for this text here,
"Learn Flutter the fun way,"
we could replace TextStyle here also with GoogleFonts.

For that of course, we must import package:google_fonts
and there the google_fonts.dart file.

And then here, maybe also use the Lato font
or whichever font family you want to use.

Now this just can't be const anymore,
and instead, it's just the color
that can be set to a const value now
because these GoogleFonts objects
which are created
with these font family constructor functions
are actually not marked as const constructors,
and therefore, you can't use const
in front the Text widget anymore.

So with that, if you force a reload,
you now also see a different font family being used here
for this text on the starting page.

61. Passing Data via Functions Across Widgets

So now that we added the Google Fonts package

and we changed some fonts here, it's now time to make sure that as we navigate through these questions, we also save the answers that were chosen by the user, so that we can then later use them on the results screen.

And of course, we should make sure that we stop once we run out of questions so that we don't see this error here.

Instead, it is indeed in this case here that we ran out of questions, that we exhausted all questions, where we want to show the results screen instead of trying to load the next question.

So therefore, we have two main things to do, we must store the selected answers and we must make sure that we show a different screen once we went through all the answers.

And the question is how do we do that?

Well, we got our QuestionsScreen widget here with its state class.

And in there, we have this answerQuestion function, which is executed whenever we click one of our answer buttons.

Here we connect this button to the answerQuestion.

Now, here we then increment the currentQuestionIndex, but it's also here where we should store the selected answers and make sure that we don't continue incrementing this QuestionIndex until we run out of questions.

And therefore, I'd say it's the quiz.dart file with its Quiz widget at the _QuizState that is important for us here.

Because for one, in here,

we manage which screen is currently shown,
and therefore, that should also be the class
where we make sure that the result screen,
which we have yet to add, is shown once we're done.
And this also should be the place
where we store the selected answers
because it's the `_QuizState` class
that has access to both the questions-screen,
which is where the questions and answers are presented,
and soon also to the results screen,
which is where the selected answers will be needed.
So we should lift state up
and manage the selected answers state
in this `_QuizState` class,
since this state class here has access to the screen
that generates the answers
and soon also to the screen that needs the answers.
So therefore, here, it makes sense to add
a new piece of state, a new variable,
to this state class here.
And this variable here can be named `selectedAnswers`.
And this should be a list of the answers
that have been chosen by the user.
And the first answer that's about to be added to this list
should be the answer chosen for the first question.
The second item that's added to this list
should be the answer that was chosen
for the second question, and so on.
Now, therefore, of course, we want to make sure that here
we have a list of string values,
hence, we should explicitly set
`selectedAnswers` to that type,

so that this here will be an empty list initially,
but will be a list of strings
once we start adding elements to that list,
which we'll do soon.

And we can also set this to be a final variable
because, indeed, I don't plan on reassigning this variable.
Instead, I plan on adding elements to the existing list,
and as mentioned a couple of lectures ago,
that is indeed something you can do with lists and objects.
You can manipulate them in memory,
final only complains if you try to assign
a new value to this variable, which I won't do,
I will just edit the existing value in memory.

But of course, initially here, it's empty,
and we, therefore, must now add a method
to this `_QuizState` class that is able to add answers
to the `selectedAnswers` list.

And for that here, we can add another new method
to this `_QuizState` class,
which could be named `chooseAnswer`.

The name is up to you,
but here I expect to get the selected answer,
which should be of type `String`.

And then inside of this function body,
I want to update my `selectedAnswers` here.

And for that, we can reach out
to the `selectedAnswers` variable here,
and call the `add` method on that.

Now, that's another built-in method,
which is available by default in Dart on all lists.

And `add` does what the name implies,
it adds a new item to the list

without reassigning that variable,
instead, it reaches out to that list object in memory
and adds a new item to that object,
which is why we can do it despite this being final,
because we're not reassigning the variable,
we're just editing the existing list in memory.

And here it's the answer

which I want to add to `selectedAnswers`.

So now, with that,

we have a way of storing those selected answers.

But of course, now we must find a way
of ensuring that `chooseAnswer` gets executed
whenever an answer gets picked
in the `QuestionsScreen` widget.

Well, and to achieve this,
we have to do what we did before in this course already.

We have to pass `chooseAnswer`
as a value to `QuestionsScreen`.

So here we can use `chooseAnswer`
by its name, without parentheses,
to pass a pointer to that function to `QuestionsScreen`.

And in `questions_screen`,
we now must accept this function.

For that here in the `QuestionsScreen` widget,
and that's important,
we're now in the widget class, not in the state class.

So in this widget here,
you can add a new final variable of type `void Function`,
that actually now does take an argument,
an answer of type `String`.

And you can name this function variable here,
`onSelectAnswer`.

This name is up to you,
but I'm using this way of naming it, `onSomething`,
to make it clear that this in the end is a function
that's triggered upon a certain event.
I'm using this because here
I also want to use a named argument again,
and therefore, here I can set this named argument like this,
required this.`onSelectAnswer`.
And using this on naming here
simply makes it a bit clearer
that here a function should be passed in
as a value without having to look at the type.
Because for all these built-in widgets,
like these buttons, Flutter also uses this convention
of naming the arguments that want functions, `onSomething`.
And that's why I'm also doing this myself here
for this named argument where I want a function.
And I want a function of this type.
A function that returns nothing,
but that takes one positional argument,
which is of type `String`.
So here in `quiz.dart`, I now have to pass `chooseAnswer`
as a value for this named argument.
So here we can now set this `onSelectAnswer` argument,
which we just added, and set this equal to `chooseAnswer`.
Now we have to get rid of `const` here
because that's not supported anymore.
But with that we're forwarding this function here,
which is defined in `_QuizState` to `QuestionsScreen`.
Now, in `questions_screen`,
we now have the problem, though,
that `onSelectAnswer` is available

in this QuestionsScreen widget class,
but we actually need it in the state class.
Because since this is a stateful widget,
it's split out across two classes, as you learned.
So how can we get hold of a property
that belongs to the widget class
inside of this state class?
Because it is in the end here in this answerQuestion method
inside of _QuestionScreenState
where I want to execute onSelectAnswer.
Well, for that, Flutter gives you a special widget property,
which you can use in this state class.
So in that class that extends State.
This instance variable called widget,
so this property called widget,
is provided by Flutter,
it's built into this state class.
And you can, therefore, also use it
in your class that extends the State class.
And this will give you access to the widget class
and its properties.
So for example, to onSelectAnswer here.
And therefore, here inside of answerQuestion,
besides updating QuestionIndex,
we can now call widget.onSelectAnswer here,
and pass our answer to this method.
Now, for that, of course,
we need to find out what our answer is.
And for that answerQuestion in here
should also accept this selectedAnswer,
so that we can then forward it to onSelectAnswer.
And therefore, in the end,

to this `chooseAnswer` method here in the quiz widget,
in the `_QuizState` class of that quiz widget.

So we have a chain of functions here

that needs to work together

to make sure that the selected answer is passed

to the top-most widget that needs it

when one of these answer buttons was clicked.

Now, at the moment, `answerQuestion`

doesn't get this string here, though.

We want to get a string here,

but in the place where we're using `answerQuestion`,

it won't get this string.

Because `onTap` on the `AnswerButton`

actually wants a function that gets no argument.

And `answerQuestion` now does get an argument.

So here in this place to make sure that we do pass

a string value to `answerQuestion`,

we should actually replace this pointer

at `answerQuestion` here with an anonymous function,

that since instead passed as a value to `onTap`.

And in here, we do indeed have access to the answer

for which this `AnswerButton` was rendered.

Because, keep in mind,

we're going through all the answers here,

and then we're creating one `AnswerButton` per answer.

That is what `map` does here, in the end, as you learned.

So therefore, here, we have access to the answer

for which this button is created.

And in this anonymous function,

which is created on the fly here,

as part of creating this `AnswerButton`, so to say,

we can, therefore, call `answerQuestion`,

and pass the answer here to answerQuestion.

And this will be different for every button.

So it will be just the answer that belongs to the button
when that button is created as part of this map function.

So we make sure that it's really the right answer
that belongs to a selected button when a button is clicked.

Now, this might look like we are now again

executing a function immediately

when before we actually passed a pointer

at a function here as a value to onTap,

but that's not what we're doing here.

Instead, we're creating

this anonymous function on the fly here,

and it's this anonymous function

that is forwarded as a value to onTap.

And that function body inside of this anonymous function

will only be executed when that anonymous function

is triggered from inside AnswerButton.

So the code here inside this anonymous function

does not execute immediately when this button is created,

but only when this anonymous function

is triggered from inside the button,

which here will only be the case

when this ElevatedButton is tapped.

So at that point of time when that button is tapped,

answerQuestion will be called,

and the answer of the button that was tapped

will be passed as a value to answerQuestion.

And in there, we then kind of forwarded

to the parent widget of the QuestionsScreen

by executing onSelectAnswer,

which was this function we receive as an argument

in the QuestionsScreen widget.

And that in turn is this chooseAnswer function here,

because that is what we're passing as a value

for onSelectAnswer to QuestionsScreen.

Now, this combination of widget and chain of function calls

can be confusing at first, but in the end,

we're just passing functions as values around,

ensuring that we're executing the right function

in the right place, and we can still pass data

between those different levels of widgets.

And with that, with every question that we answer,

we'll add a new answer to this selectedAnswers list here.

62. More Conditions

So with that,

we're storing all these answers,

but now we also must make sure that at a certain point

of time we don't continue presenting questions

because otherwise we'll run into an error again

once we run out of questions.

And to make sure that we don't reach this page,

we can actually use this selectedAnswers list here

to find out at which point of time

we have actually answered all available questions.

Because of course here, we add one answer per question

to selectedAnswers.

Initially it's an empty list

and at some point it will have as many answers

as we have questions in this app.

Therefore, what we can do inside

of this chooseAnswer question here

is we can add a if statement

about which you learned a little bit earlier already,
and compare the length of selectedAnswers
to the length of questions.
And if the two are equal, we know that we have answered
as many questions as we have.
So it's then at this point of time where we should stop.
So to make that comparison here,
we can access selectedAnswers.
And then on this list object here,
we got various utility properties
and methods as you learned.
For example, we have a length property
which gives us the amount of items stored in that list.
And we can check if that is equal to questions.length.
Now, for this, we must make sure
that the questions defined in this questions.dart file
are available in this quiz.dart file.
And that's of course done by importing them,
by importing from package:adv_basics.
And then there from the data folder
and there from questions.dart.
With that, we import this list of questions from that file.
And in quiz.dart, we can now compare these two lengths
and if they are equal, we know that we ran out of questions.
So that's when we wanna switch to the results screen.
At the moment, I have no results screen though,
therefore, right now I simply wanna switch
back to the start screen.
Hence, in here we can call setState
to trigger the build method again.
And in this anonymous function which we pass to setState,
we can set the activeScreen back to startScreen

to make sure that we show the startScreen
instead of an error once we ran out of questions.
With that, if you save this and force the app to reload,
you should be able to start the quiz
and answer these questions by picking any answers.
But once you're done,
you're back here on the starting screen
instead of getting that error, which we got before.
Now what you'll notice is
that if you now start the quiz again
and you go through all questions,
you will get this error screen again.
And we get this error
because we don't reset our selectedAnswers.
Therefore, we'll actually have more answers than questions.
And this condition will never be met again
after it was met for the first time.
And we could work around that
by simply resetting our selectedAnswers
back to an empty list.
For this, we now just have to switch from final to var,
or since I'm defining a type here,
actually just remove final since this will then
automatically turn this back to a variable.
Because we can only set this to a new value
with the equal sign operator if this is a variable.
That's what I mentioned before.
So with this switched back to a variable,
once we exhausted our questions here,
we should reset selectedAnswers back to an empty list.
And with that, if you now save this
and force a restart, you can start the quiz again,

go through all questions.
And if you now start again,
you can go through them again without running into an error.
Of course, that's not the final version of the app.
The results screen is still missing,
but that's how we can use this `selectedAnswers` state
to show a different screen
once all questions have been answered.

63. Getting Started with the Results Screen

So now that we have this starting screen
and then the questions screen with all these questions,
it's time to also add a results screen,
which should look like this.
The goal with this screen is to simply show a summary
of all the questions that have been answered.
The user should see how many questions
have been answered correctly,
and then they should simply see a list
of all those questions
with a clear indicator that shows
whether a question has been answered correctly or not,
and with the correct answer next to the answer
that has been chosen by the user.
This list of questions here
also should be scrollable
since it shouldn't occupy the entire screen,
but instead it should be placed
between this summary statistics title
and a button that can be used to restart the quiz.
So that's what we're going to work on now.
And for that, as a first step,

we'll add a new file for a new widget,
and that, of course, will be our results screen widget,
which will be stored in our results_screen.dart file.

Now, this results screen widget
will actually be a stateless widget
because it won't need to manage
any internal state as you will see.

So this is my ResultsScreen class,
which extends stateless widget as you learned.

And of course, as always,
we therefore, must import this Flutter package
and the material.dart to unlock stateless widget.

And in this results screen widget,
you then should add this build method as always.

And in here, I want to output this screen,
which I just showed you a couple of seconds ago.

Now, if you take a look at this screen,
you, of course, see that there are a couple of items
placed on top of each other, above each other.

And that always is a clear indicator
that we want to use a column
because as you learned already,
this is a widget built into flutter
that can be used to display a vertical list of widgets.

Therefore, in this build method, in the ResultsScreen,
we can return the column widget
and set its children argument
to a list of child widgets
that should be inside of this column.

Now, just as in the questions screen,
my column should take up all the available width though,
and it, it should have some margin around it.

And therefore, we can indeed use that same pattern
that we already used in this questions screen widget.
Hence, I'll copy that code
and in my results screen,
I'll put that before my column here,
and then make sure that my column widget
is actually set as a child inside of that container widget.
And then I need a couple of closing parentheses thereafter
so that the code looks like this.
With that, I have a column widget here,
which generally takes up the entire space on this screen.
And just as on the questions screen,
I then also wanna make sure
that I center my items along the main axis.
So therefore, I'll also copy this setting over
to the column widget in the results screen widget.
And now we can get started adding some widgets here
to this list of widgets.
And here the first widget
is actually that text widget
that tells us how many questions
have been answered correctly.
And for the moment, I'll just add
some hard-coded dummy text here
where I'll say, you answered X out of Y questions correctly.
And this can then be turned into a constant value
or actually the entire list can be turned
into a constant value right now.
So with that, we have this text here.
Then I want to add a sized box
to add some vertical spacing
by giving this a height of let's say, 30.

You can, of course, always tweak this height value
if you want to.

And then below that, I want to have
that scrollable list of questions
where we have the question, the correct answer
and the chosen answer.

So I'll get back to this.

I'll just add a placeholder text widget for now,
list of answers and questions.

And below that list,
which will be added here in the future,
I want to have another sized box for some spacing,
and then that button that can be used to restart the quiz.

And for that here, I'll just use a standard text button.
Of course, you could also use some other kind of button,
but here a text button it is.

And that text button takes the onPressed argument
and a child argument.

Now the child argument should be a text widget,
which sets the text that is displayed on that button.

And in this case, the text should be restart quiz.

And for onPressed,

I'll just add a dummy empty function for now.

And here I am getting an error
because of the const keyword in front of this list here,
because all these widgets can be set to constant values,
but the text button can't.

Therefore I'll just manually add const
in front of all these other widgets,
but not in front of text button.

But with that, of course,
this button doesn't do anything right now,

but we'll soon change this.

This is just the general structure
of this customer ResultsScreen widget.

To complete this widget for now,
we also must add our constructor function here
and accept and forward that named key argument,
and this can also be turned
into a const constructor function.

Now, with that, we can add some logic
to show this result screen
instead of that starting screen
once we exhausted all our questions.

So here in the quiz state class
where I currently set the active screen to start screen
once we exhausted the questions,
I wanna set this to results screen instead.

And then in the build method here,
I, of course, also wanna set my screen widget
to that results screen
if active screen is set to results screen.

And for that, we first of all, have to import
this results screen dart file in which we just worked
to unlock this results screen widget in this file.

And then here in this build method
of this quiz widget of this quiz state class, to be precise,
I wanna check if my active screen variable
holds a value of results dash screen,
which, of course, is the value we are setting here.

And if that is the case,
I'll set screen widget equal to ResultsScreen like this.

And then we can add const in front of this.

And if you now save this and force a reload,

you should be able to start the quiz
and go through all these questions.
But once you answer the last question,
you end up on this dummy ResultsScreen,
which is not finished yet,
but which generally shows the content
we just added to that screen.
Of course, now we must make sure
that this button here works if we press it
and we need to output
the actual summary information for this quiz.

64. Passing Data to the Results Screen

Now with the results screen widget added,
it's now time to make sure that we pass our selected answers
to this widget so that in there we can use them
to show some results on the screen.

And as always, if you need to receive some data in a widget,
if you wanna be able to pass some data into a widget,
you must accept an argument here
in the constructor function.

And then in order to, for example,
use it in the build method,
you also must add an extra property to this class.

So here in results screen,
I'll add a new final property called chosenAnswers,
for example, which is of type List, to be precise,
a list full of string values.

And now here in this constructor,
we can either add a named argument
or a positional argument.

That is up to you.

I'll go for a named argument here
and I'll therefore add
a named argument called `chosenAnswers`
and map this to the `chosenAnswers` property
by using this `this shortcut`.
And since named arguments are optional by default,
I add `required` in front of it to make sure that a value
for this argument must be provided
when creating this widget.
With that, in `quiz.dart`,
we now must pass our chosen answers to `ResultsScreen`.
So we must pass our selected answers list to `ResultsScreen`.
And of course, that's fairly easy to do
because in here we already have the `selectedAnswers` list,
so we can simply forward this variable,
or to be precise, the list stored in this variable
as a value to `ResultsScreen`.
And we pass this list to `ResultsScreen`
by targeting the `chosenAnswers` named argument
and setting it equal to `selectedAnswers`.
Now we have to get rid of `const`
since `selectedAnswers` is a list
that is dynamically changed in our code.
It's a variable, after all,
that is set to a brand new empty list
at a certain point of time once we exhausted all questions.
So it's variable
and therefore this `ResultsScreen` object
can't be marked as constant.
But with `const` removed, this works,
and we do now receive our chosen answers
inside of `ResultsScreen`.

There we can now use it
to output some real summary statistics data here.

65. Introducing Maps & "for" Loops

Now in order to output some useful
information in this results screen here,
we now have to kind of combine the chosen answers
with the questions that we have in this separate
questions.dart file.

And for that, I'll add a new method here to results screen
and I'll introduce you to a new data type.

But let's start with the method.

I'll add a method called get summary data.

The name of course as always, is up to you.

And in this method here, I wanna return a list of values,
but now not a list of strings or a list of widgets,
but instead a list of so-called maps.

And that's now not a map as in Google Maps,
but instead in programming, a map is a very simple data
structure that maps values to keys.

And in dart, map, just like list, is actually a generic type
where you specify the type of your keys
and the type of your values.

So we need another pair of opening
and closing angle brackets here.

Therefore, we have some nested angle brackets here,
which is absolutely something you can have in dart.

And this inner pair of angle brackets now belongs to map.

And for my keys, I'll have a bunch of strings.

So the keys in my map will simply be string identifiers,
but for the values, I'll have different types of values.

I will have strings and also numbers

for these different keys.

And therefore here, the second type I specify for this map generic type after this comma here, which is the type of values that will be stored in this map should be object written like this with a capital O.

Since all values in dart are objects at the end, this is a pretty flexible type that actually allows all kinds of values.

And this allows me to store different kinds of values in this map here.

So that this map can simply be used as a data structure that groups different kinds of key value pairs together.

Alternatively, we could create a brand new class that defines how our summary objects should look like.

And then we could instantiate a bunch of summary objects here

and that would definitely also be a great approach.

But I also want to introduce you to this map value and type, and that's why I'm using this approach here.

So get summary data should return such a list of maps.

Now to generate this list here, I'll start with an empty list, which I'll store in a variable or in a final variable to be precise, called summary.

And the type of this list here should be this type, since in the end I plan on returning summary here.

But of course in between creating it as an empty list and returning it, I also wanna populate that list.

And for that, we'll use a new dart feature which we haven't seen before.

We'll create a for loop.

Now, early in this section we saw if statements to run some code conditionally.

And if statements are so-called control structures because they allow you to control the code execution flow.

Another kind of control structure supported by dart and many other programming languages as well, are for loops.

With the four keywords, you create such a loop and a for loop simply allows you to execute a certain code snippet multiple times.

To do that, you also add parenthesis here.

And between these parentheses you add your for loop configuration, so to say.

This configuration consists of three parts.

The first part is a helper variable that's created for this loop.

Typically this variable is called `i`, but the name is up to you and it's set to an initial value.

Very often that value will be zero, but any value can be used here.

Then thereafter you add a semicolon, so not a comma, but a semicolon,

which might look like a strange place since thus far we always use semicolons at the end of lines.

Now we're using it here in the middle of a line.

But that is just how the for loop must be defined.

And we continue with the next piece of configuration for the for loop.

And that next piece of information that we have to add here is a condition that controls how long this loop should keep on running.

For that, you typically use this helper variable here, `i`.

And here, I want to keep on running as long as I'm still processing chosen answers.

So as long as `i` is smaller,

and it's a new kind of comparison operator,
which you haven't seen before,
but just as two equal signs can be used to check
for equality, smaller can be used to check whether
this value on the left side is smaller than
the value on the right side.

And the value on the right side here should be my
chosen answers.

And there the length of this list,
which also will be a number, just like I.

So now with this code, the loop here, which we are about
to define, will keep on running as long
as I is smaller than the length of my chosen answers.

So if I have six answers in this list, this will be true
as long as I is smaller than six,
which it initially will be since we initialize it to zero.

And therefore we need a third part here
when configuring this list, where we actually control
how I should change after every iteration.

Because if it never changes, we create an infinite loop
since I then would always be smaller than the length
of this list.

Therefore here, you then control how I should change.

And typically you increment I by one
by using this ++ shortcut here.

You could also set it to I+1 like this manually,
but this is simply shorter.

And then you add opening and closing curly braces.

And between those curly braces,
you have your loop body, you could say.

And that will be the code that will be repeated
until this condition here is not met anymore.

And in here, between these curly braces,
I now want to add items to this list here.
So for that, I reach out to this summary list
and call the add method to add items to it.
And as you learn, you can do this on final lists.
Just reassigning this to a new value wouldn't work,
but calling add will work.
And the value I wanna add here is such a map.
So such a simple collection of pieces of data
of key value pairs to be precise.
And in dart, such a map is created by using opening
and closing curly braces.
Now of course, we have also already used these curly braces
a couple of times in other parts of the code, for example,
for this loop body or to mark the body of a function.
But it can also be used in places where a value is expected,
like here for this add method to create a map
because this is the dart syntax for creating a map value.
Now, between those opening and closing curly braces,
you then put your different key value pairs.
And here you can add any keys of your choice
as long as they are of type string.
So for example, we could add a key named
question_index and then set this equal to a value
by adding a colon, not an equal sign.
Inside of a map, you use a colon
to separate the name from the value, for variables
you use an equal sign, in a map it's a colon.
And on the right side of the colon, you set the value
for this key in this map.
And here for question index, my value would be I.
Because I'm going through all my answers,

and as mentioned earlier in this section, we'll have
as many answers as we have questions.

So implicitly with that loop, we're also going
through all the questions, kind of.

And I will change after every iteration,
so therefore it will initially be zero,
and for the next iteration it'll be one
and then two and so on.

And therefore I here can be used to reflect the index
of the question for which we're adding some summary data
to this list here.

But of course the question index is not all,
hence you should add a comma.

And then thereafter, the next key value pair
that should be added to this map.

And the next key could be question
to add the actual question text to this summary object.

And I'm creating these summary objects so
that thereafter I'll have an easier time outputting all
these summary statistics.

So here the question text is next.

And for this we can now access the questions,
so not the chosen answers, but the questions
that are stored in this question dart file.

And to access this, we of course have to import this file.

So in this results screen dart file,
we import package advanced basics
data.dart as we did in many other files as well.

And with that here for this question key,
and I'll add some commas to get proper formatting.

So for this questions key, I wanna set this equal
to questions, but then not to the entire questions list,

but one specific question text.

So for that, I access a specific question by its index

I, which is in the end based on my chosen answers.

But since we populate these answers based on our questions

in the questions screen in the end,

I can also safely access my questions with this index here.

And then of course, I want to get the text here

for a question.

Because this question key should store the question text,

then I also wanna store the correct answer.

And that's now why it was important

to not shuffle the original answers list.

Because now I rely on the fact

that the first answer in here is still the correct answer.

So that here in results screen,

we can set correct answer equal to questions I,

for the current question, for this current iteration,

.answers and then zero.

Always the first answer because as mentioned,

these answers in these dummy questions are set up such

that the first answer is always the right one.

And last but not least, I want to add one last piece

of information to this map.

So to this data collection here,

and that would be the user answer,

so the answer chosen by the user.

And here, we want to access chosen answers

and then the index I because we're looping through

all the chosen answers, so step by step we get

the different chosen answers.

And with that, after we went through all the chosen answers,

we'll have a summary list full of such map objects,

which we can now use to output our
summary statistics down there.

66. Accessing Map Values & Using "Type Casting"

Now with the summary data being created here,
we can use that summary data to output it here
instead of this dummy text widget.
Instead, here I again wanna display a Column widget
with one row per summary data,
so to say per map in this summary list.

And then in every row,
I wanna display the question index,
some indicator that shows us
whether the question was answered correctly or not,
and I wanna display the question,
the correct answer,
and the user answer.

We could do this all here
in this ResultsScreen widget,
but this widget tree here
would get rather big and complex.
And for that reason, since, as you learned,
it is considered a good practice
to break your widgets apart
and split them across multiple smaller building blocks,
I'll do just that.

I'll add a new file here,
which we'll name questions_summary.dart,
and in there I'll add my QuestionsSummary class,
which extends StatelessWidget,
and that's one "s" too much.

So StatelessWidget is extended here

because I won't need to manage any state in here.

And as always,

we should import from the Flutter package

the material.dart file

to unlock StatelessWidget.

Now inside of QuestionsSummary,

we of course need that build method,

as we always do.

And I also need to accept some data because,

of course, I want to use QuestionsSummary

inside of ResultsScreen

in place of this placeholder Text widget here.

And I of course want to pass my summary data.

So there's a list of maps to QuestionsSummary.

Therefore, in QuestionsSummary,

I'll add a new property,

which should be a list

that contains all these maps.

These maps where the keys are of type String

and the values are any kind of objects.

And the name of this property could be data

or summaryData to be a bit more specific.

Now, of course, just adding the property is not enough.

Instead, we need this QuestionsSummary constructor function

to accept and set this summary data.

And here for that,

I'll accept a positional argument.

Though, of course,

you could also use a named argument,

as we did it before for our widgets as well.

And I'll add at least one named argument.

I'll add this special key argument,

which is forwarded to the parent widget,
and this can be turned into a constant constructor.
With that, we get and set our summary data here,
and we can now use that here to output that Column,
which I mentioned before,
where every row should represent
one piece of summary data,
so one map in this list of summary data.
So for that, back in questions_summary.dart,
we have to set the children argument of Column.
And before, we always set this to a list
where we then manually added all the widgets
that should be displayed on top of each other.
Now I will not do that here, though,
because I already have a list here,
this summaryData list.
But of course, that is a list of maps,
not a list of widgets.
And children wants a list of widgets.
So what can we do?
Well, we can transform this list
from a list of map objects to a list of widgets.
And we do this by using summaryData here
and calling the map method on that list.
And by the way,
the fact that this method is called map
has nothing to do with this Map value type;
map simply means that we map a value
to a new value.
So here we pass our function to this map method,
and this function, as before,
receives an argument automatically

passed in by Dart.

In this case, it will be one piece of data,
you could say; this name is up to you.

And in this function,

we then wanna return the widget
to which this piece of data should be mapped.

And the widget to which I want to map it here
should actually be a Row widget.

And that's now a new widget
which we haven't used before.

The Row widget is, in the end,
the horizontal equivalent to the Column widget.

Where Column is used to display multiple widgets
next to each other vertically,

Row is used to display multiple widgets
next to each other horizontally.

Therefore, just like Column,
Row takes a children argument,
which we set to a list of child widgets.

Now here you see that I'm currently getting an error,
though, and the error I do get
is that my Iterable thing here
can't be assigned to a parameter of type List.

And I get this error
because map actually returns
a so-called Iterable object,
which is almost the same as a list,
but not entirely the same.

And this is just some internal thing in Dart,
but thankfully, such an Iterable object
can easily be converted to a list
by calling `toList` here at the end.

So we chain the toList method after calling map.

So on the result produced by map,

which is this Iterable object, we call toList.

And now here we generate a list of rows.

And in every row,

I then wanna output various pieces of information.

For example, the index of the question

for which we're outputting data here.

And here we can start quite simple

and simply output a Text widget for that.

And the text that I wanna display here

is my question index plus one.

Because as you might recall,

these indexes start at zero;

list indexes start at zero.

But if I wanna display the number of a question

to the user,

it would be rather strange if I start at zero

because we humans tend to start at one

for the first question or the first item

in a list of items.

That's why I want to add one to that index.

And I can do this by accessing data

and then the 'question_index' key,

which is accessed like this.

So you access values in your maps

just as you access values in lists

by using these square brackets.

And then between those square brackets,

the key of the value

that you want to get access to.

For lists, it was the index

which you put between those square brackets.

For maps, it's the key of the value instead.

So between these square brackets,

you then use one of the keys

you defined when creating the map.

So 'question', 'question_index', 'correct_answer',

or 'user_answer'.

And here I'm using 'question_index'.

This then provides the value

that's stored under that key in this map.

So for 'question_index',

it provides this question index,

which should be a number.

I then want to add one to it.

And this now causes a problem here.

It actually causes multiple problems, as you will see,

but it causes one main problem

because here Dart does not know

which kind of value will be stored

under that question key

because I set this map object to be an object

where the values can be of all kinds of types.

Therefore, Dart does not understand

that here this i value is a number.

I mean, it knows here that it's a number,

but for the final map which it produces,

it just knows that all these different keys

can have all kinds of value types.

It does not memorize the exact value types

for the specific keys.

And therefore here,

this value returned by accessing the question key

is also considered to be of any kind of type.

Therefore, here we have to do something
that's called type casting.

You can use the `as` keyword, which is built into Dart,
to tell Dart that you know more than Dart knows
and you know which kind of value this will be.

You don't have to use type casting a lot,
but when working with maps
that store mixed kinds of values,
as we're doing it here,
it's quite common that you need to use type casting.

After `as`, you specify the type of value
you know you'll get here.

So in my case here, an `int`.

Now this should be wrapped into regular parentheses:
an opening parenthesis before `data`
and a closing parenthesis after `int` here
to tell Dart that this should be evaluated as one value,
so to say.

With that,

Dart knows that this here will be of type `int`,
and it then adds one to it.

But now we have a new problem,
and the new problem is that we can't pass an argument
of type `int` to the `Text` widget
because that actually wants a value of type `String`.

Hence, we must convert this number here to a string.

And one way of doing that
is to wrap this with another pair of parentheses,
but now the closing parenthesis after this plus one.

And then on this overall value that is derived here,
which is an `int`,

you can call toString to convert this int to a string.
So a bunch of parentheses,
which we here also have to use
because we're working with such a map,
which is not very precise type-wise.
But I of course also wanted to show you
this Map data type in this course
because you will encounter it from time to time
in your Flutter apps,
and this is how you can handle it
and how you also can convert that int value to a string.
And with that,
we should be outputting the question number here
as a first item in this row of widgets here.
But of course, that's not all that should be output.

67. Combining Columns & Rows

Next to this question identifier here,
still in that row here though, I wanna output another column
and that's also something you can do
and you will do quite frequently with flutter.
You have rows inside of columns
which then might contain more columns
so that you can restructure the content on your screen
exactly as you want to structure it.
And if in a row, next to another item,
you want to have a couple of rows,
you need a column inside of a row
because a column widget allows you
to display multiple items on top of each other vertically,
which is the definition of having multiple rows.
So here I'll add another column so that inside

of this column I can output the question,
the chosen answer and the correct answer.

So for this here in this list,
which again should be a list of widgets,
we can start with the question
output with help of the text widget.

And for this we can access data question.

And this here actually should be question index, of course,
therefore, so here I want to access the question.

And again, dart does not know which kind of value
we'll get here,

but it does know that it wants a string
and this potentially isn't a string.

It could be any kind of value.

And therefore we again have to do some type costing here
to tell dart that we know that this here will be a string.

And with that we're outputting that question.

Now below this text here, which represents the question,

I want to have a little sized box for some spacing,
some vertical spacing by setting some height here,
and then more text widgets,

one for the selected answer and one for the correct answer.

Now for the selected answer,

we can and should access our map.

And then they are the user answer

because that is the answer chosen by the user.

So user answer it is here.

And again, we need to tell dart that this will be a string.

And then below that we output the correct answer

by accessing correct answer on this map here.

And with that, we should be outputting

a list of summary data.

So therefore now we can go back to the results screen
and use this questions summary widget there.

So for that, as always,
it must be imported there from our own package,
the questions summary dart file,
and with the import added, here instead of that placeholder,
we can now output questions summary.

And we have to pass our summary data to questions summary.

And that simply means that we must execute this function
which generates and returns that summary data.

So therefore down here we should call get summary data
and execute it as a function.

Here we're not passing the function name as a value
because questions summary does not want
a function as a value here, but instead a list of maps.

And that's exactly what gets summary data produces,
but therefore we also have to call it here.

And with that, if you saved this all and reload everything,
you should be able to restart the quiz
and pick any answers here.

And you end up on this results screen
where we actually got no list here.

What's the problem here?

Well, the problem is this line of code
in our quiz state class in the quiz dot dart file.

Here we are resetting the selected answers
back to an empty list.

And therefore, of course, since you call set state,
when the build method executes,
the next time we're passing this empty list,
which we set here as a value to the result screen.

So there basically are no selected answers there.

Now we did this before
to avoid an error when we restarted the quiz,
and we will still have to do that when we restart the quiz,
but now we aren't restarting the quiz here,
we're showing the results screen instead.
So therefore here this code should be removed
and we'll bring it back later once we take care
about adding a restart quiz functionality.
For now, we should get rid of it.
And therefore now if we save this and restart,
we should be able to answer here.
And now we have our list here.
Of course, pretty ugly, no formatting.
Also in my case here with some strange error,
which is simply a little feature
provided by flutter when running this app in development
to show you that some widgets are cut off
and are going outside of your screen boundaries,
but we'll take care of all of that soon.
What you can still see
besides this being broken,
is that we have our question numbers here
and that in the end we seem to have some rows here.
For example, this here is a row,
this here is another row.
And in every row we first have the index here,
and then next to it we have more columns, here for example,
where we then have the question and the chosen answer
and the correct answer.
So our layout generally works.
We have an overall column with multiple rows,
and in every row we have two columns,

one column with the index,
and then another column with more rows
with the question and the chosen answer
and the correct answer.
But of course, as mentioned, it's ugly and it's broken.
So that's what we'll fix next therefore.

68. Expanded To The Rescue!

So we're outputting
some basic summary data here,
but of course at the moment there are these ugly warnings
that some content does basically not fit
into the screen boundaries, and that's of course not good.
To fix this, you should go
to the questions summary.dart file
and to the questions summary component there.
And there you can fix this issue by wrapping this column,
which is inside of this row,
so not the outer column,
but this inner column with another widget.
For that, we can refactor this of course
and wrap it with a widget.
And the widget
with which this should be wrapped is the expanded widget,
which is a widget we haven't used up to this point.
Now I'll explain it in just a second.
If we save this,
you should see that now this is reformatted here
and now text that is too long
for a single line instead is broken across multiple lines,
which is better.
The overall styling is still not great,

but this is a good start.

But what exactly does expanded do here?

Now, the name expanded here might sound

like the child widget takes all the space that it can get,

and to some extent it does,

but that's actually better than what we had before

because without expanded,

column actually takes an infinite amount of width,

even if that goes beyond the screen boundaries.

After wrapping it with expanded, this changes,

because expanded basically allows its child

to take the available space

along the flex widget's main axis,

which might sound strange,

but the flex widget is simply a row

or a column widget above the widget

that you put into expanded.

So the flex widget

for this column here would be this row widget, for example.

Now the maximum width

of a row widget is the maximum screen width,

just as the maximum height

of a column widget is the maximum height of the screen.

Therefore, if you use expanded inside of a row,

the widget that is passed as a child

to expanded can't take more than the width of that row.

That's what expanded guarantees.

Without expanded, the column inside

of the row totally ignores the maximum width of the row

and instead grows beyond that width.

With expanded, it's restricted to that width of that row,

and this column here will instead only expand

such that it fills up the space provided by row.
That's why in the end,
expanded here restricts the width of the column
instead of increasing it.
It restricts the width of the column
to the width of the row,
which in the end is the width of the device screen.
Of course, we'll also soon take care
about the rest of this styling.
Before we do that though,
I want to add these summary statistics here
with concrete values for X and Y.

69. Filtering & Analyzing Lists

Now in order to populate this area here
with the correct data,
so the number of correctly answered questions,
and the number of total questions,
we have to go to the results_screen.dart file.
And there in the build method, we have this text widget,
where we wanna output the number of total questions
and the number of correctly answered questions.
How do we now plug in the correct values here?
Now for this we can add some final variables
to the build method.
The final because I don't plan on reassigning them
inside of the build method.
But I'm adding some variables here
to dynamically derive these X and Y values.
And whilst we could do this here in place,
injected into this string,
it would quickly lead to quite unreadable,

or at least hard to read code.

And I'm doing this up here at the beginning
of the build method to keep my overall code readable.

So here I want to derive the number of total questions,
which could be stored in a variable called
numTotalQuestions,

which I'll temporarily set to a dummy value of zero,
but this will change soon though.

And then I want to have my number of correct questions,
which I initially also set to zero,
but of course now zero should be replaced
with the actual correct numbers.

Now for numTotalQuestions, that's simple.

We are importing our questions here.

So in the end, this questions list here,
defined in the questions.dart file,
and therefore we can simply set numTotalQuestions
equal to questions.length.

Because that is our total amount of questions.

Now to find out which questions were answered correctly,
we have to use our summary data here.

Because in that summary data,
we do store all the chosen answers and the correct answers.

And if we compare these two values,
we can find out whether the correct answer was chosen.

So therefore, here in the build method,

I, when I get my summary data,
and for that I'll add a new variable called summaryData,
which stores the result of calling getSummaryData.

And I'm doing this here because I'm also using the result
of calling this function down there,
and to make sure that we don't unnecessarily call

the same function twice within the same build method,
I will call it once here, store the value in a variable,
and then reuse that value by simply using that variable,
which is one of the advantages of using variables.

So with that, I'm using that summary data down there,
but I cannot also use it here to derive the number
of correctly answered questions.

To do that, we can use this summary data,
which is a list of maps where every map looks like this
to find out for how many map objects user answer
is equal to correct answer.

For this on summary data, so on this list of maps,
we can call the built-in where method,
which just like map or add, is a method provided by Dart.

Now the where method allows you to filter the list
on which you're calling it.

But just like map, it does not change the original list,
but it instead returns a new list.

Or to be precise, it also returns such a iterable thing,
which is almost like a list.

So where can be used to filter this original list
and create a new filtered list based on that original list.

And two, where you have to pass a function,
just as you pass the function to map,
which will be executed for every item in that list.

And just as with map, you will then receive that item
as a argument in that function which you pass to where.

So here I get my data point, my map,
and then inside of this function, unlike map,
you don't return some transformed value,
because that's not what where is used for.

But instead you return true

if that data point should be kept in that new filtered list
that is returned by where.

Or false if it should be dropped.

And of course the trick is to not hard code
true or false here, otherwise all values will be kept,
or all values will be dropped,
but instead to dynamically derive
whether it should be true or false.

And for that we can define a condition here,
just as we did it before this section
when we learned about if statements.

Here, I wanna dive into my data point, so into my map here,
and compare the value for the key user answer,
to the value for the key correct answer.

So therefore here I access user answer,
with that syntax you learned about,
accessing map values by keys like this.

And I compare this with the double equal sign operator
because one equal sign would try to assign a new value,
which is not what I wanna do here.

I wanna compare this value with two equal signs.

I wanna check for equality, and I'll compare it to data,
and then the correct answer.

So to this value stored under this key here.

And if these two values are equal,
we know that the user chose the correct answer,
hence this will yield true.

And this data point here will be kept in that new list
that is generated by where.

If the answers differ, then this here will yield false,
and that data point will be dropped in this new list.

And with that, we generate a new list.

But of course here I wanna have a number
of correctly answered questions,
not a list of correctly answered questions.
And therefore here at the end we should access `.length`,
which exists on these iterable objects here as well.
And this then gives us the amount of items
that were kept in this filtered list.
And with that, we now have all we need.
The number of total questions,
and the number of correctly answered questions.
We can now use this here to inject it into this string.
So to build this string dynamically
with that syntax you learned about earlier in the course,
where you can add a dollar sign in a string.
And then here we wanna have the number of all questions.
So the `numTotalQuestions`.
And `X` should be replaced with dollar sign,
`numCorrectQuestions`.
And now since we create this string dynamically,
it can't be a constant value anymore,
hence `const` should be removed.
And I'll also add a comma here at the end after this string,
so that this can be broken across multiple lines,
at least a little bit.
Well, and with that, you now see that indeed
I do have some concrete numbers here.
And with that we're outputting the right data,
but of course, it's not looking that good.
And I also wanna make sure that this part here
is actually set to a fixed height,
and is then scrollable inside of that box,
which is not the case right now.

70. Making Content Scrollable with SingleChildScrollView

So I wanna make sure that this part here
where I output the data about the individual questions
is set to a fixed height,
and then I want to be able to scroll
through the items in that box here.
At the moment, that's not the case.
So how can we set questions summary to a fixed height?
Well, for that, we can go to that widget here
to this building block.
And then in there, we wanna wrap this column
with another widget.
With a widget we used multiple times
before already the SizedBox widget.
Now I'm not using it to add some empty spacing
between two items as we did it multiple times
before in this section.
But instead, I'm using it to create a box
with a fixed height,
in this case, by adding the height argument
and the child widget that is then placed inside
of SizedBox will have to adhere to this height.
It will be cut off if it exceeds that height.
And here we could set this to, let's say, 300 pixels.
I'll actually make this a bit taller later,
but I'll start with 300 pixels here.
And you see that as I do this, I get a render overflow error
and I again get a warning
that basically my column goes beyond these boundaries
that I set here.
It's taller than these 300 pixels that I assigned.
And that's not ideal

because that means that indeed I can still see it here,
but it's now below this button, which I also have here
because the `SizeBox` essentially ends here
and everything below that `SizeBox`,
then in this case, still is visible,
but is ignored when it comes to the placement
of other widgets like this button here.

So therefore, we must make sure
that inside of this box here, we can scroll,
so that Flutter hides that content
and instead allows us to scroll to it.

And that can be achieved with help
of another widget built into Flutter.

Because Flutter is all about widgets,
as you, of course, know by now.

And it's this column inside of this `SizeBox`,
which we just added, that should be wrapped
with that other widget.

So it should be wrapped again,
not just by `SizeBox` to set this general height,
but it should also be wrapped with another widget,
which is called `SingleChildScrollView`,
another widget built into Flutter.

And this widget kind of does what its name implies,
it takes a single child.

In this case, this column,
which of course, contains multiple child widgets,
but `SingleChildScrollView` only has one direct child.
And that is that column.

And it makes that child scrollable,
so that if this child, this column in this case
exceeds the boundaries set around it,

in this case, set by this `SizeBox`, it becomes scrollable.

That's in the end

what this `SingleChildScrollView` does here.

Therefore, if you save this,

you see the content that was below

this restart quiz button is gone.

And instead you can now drag here to scroll this widget

or to scroll the content in this widget.

And that's exactly the behavior I want here

to make sure that we define a fixed size here.

And we can then scroll in this box

to still see all the content.

Of course, here, we would've had enough vertical space

to display all the content,

but we're also not done with the styling yet.

And therefore, we might take up more space once we are done.

And it is, of course, also important

to know about `SingleChildScrollView`,

because as you're building Flutter apps,

you will often have situations

where content should be scrollable.

You will later also learn about some other widgets

that help with scrolling.

But `SingleChildScrollView` is definitely a key widget

you should also be aware of.

71. Beyond the Basics: Optional, Important Dart Features

Now that we've finished this app here,

and now that we've finished

this course section here in general

and we learned about many crucial flutter and Dart features,

I wanna end this section

by diving into three Dart features you should know,
which are totally optional though,
which you don't have to use in this or other projects,
but what you might want to use
or what you might see in other projects you are working on.
And also features we will use from time to time
throughout the rest of this course,
which is why you should know about them.
And I wanna start here in the quiz Dart file.
There we have this QuizState class,
which is this state class for the quiz widget.
And you learned that stateful widgets
are basically split across these two classes.
Now the QuizState class here
is actually called underscore QuizState.
And I mentioned earlier in this section
that this leading underscore
is a Dart-specific thing in the end.
By adding an underscore in front of a class name,
you tell Dart that this class is meant to be private
and should basically only be usable inside of this file.
But if another file would import quiz.Dart,
it would not be able to access this QuizState class.
The reason for that is that this QuizState class
really belongs to this quiz class
and should therefore not be used outside
of this file in any other context.
The quiz class then also does not have
that leading underscore because it indeed is not private,
but publicly available.
Now that is not a new concept of course,
I mentioned it before,

but it is a concept you should keep in mind.

But what you should know is that you cannot just turn classes into private classes by adding such an underscore, but all the properties and methods inside of classes.

We could, for example, convert selected answers to a private property by adding a leading underscore.

Of course, we have to do that in all the places where selected answers is used thereafter.

Otherwise we have an error that some identifier can't be found.

But with that, you would make it clear that selected answers is really meant to be only used inside of this file here.

And if another file could get access to the QuizState class because it maybe would not be private, then that other code in another file should still not be able to access selected answers because even though the class is now public, this is still marked as a private property that should only be usable inside of this file here essentially.

Now here, I'll switch this back to a private class, and this is not something you have to do here. It's not important for this app we're building here, but you will sometimes see property names that start with leading underscores.

And it is important to understand that this in the end means that these are private properties or private methods, you can add such an underscore there as well.

And this simply means that those properties

and methods are meant to be only used inside of the file where the class is defined.

And if the class is the only thing defined in a file, then those properties and methods would essentially be private to that class and would not be usable anywhere else in your code, in any other file.

So that's what this syntax is all about.

Another Dart feature, which is optional, but which you should know about are getters.

And I can show you a great example for a getter here in the results screen widget class.

There we have this get summary data method, which in the end simply produces a list which it then returns.

And this list which it produces is based on this chosen answer's property in the end, that and this questions list, which we defined in another file.

Now defining this as a method like this is absolutely fine, but if you have methods like this which take no input values, no parameters, and which in the end then just produce some data based on some other class properties, you could also use a getter instead, by using an alternative Dart syntax.

For that syntax you would rename that method to sound more like a property so that it would describe data that's inside of it.

For example, summaryData.

Instead of get summaryData.

You would then remove those parentheses

and instead add the `get` keyword in front of it.

This is a special keyword in Dart, which creates a getter.

And a getter in the end is still a method.

You still have your function body here,

this code still executes,

but you don't call `summaryData` like a method.

So you don't call `summaryData` like this

with parentheses thereafter,

but instead you simply refer to it as a variable.

So now here we would have some name duplication

and creating this extra final variable here

would now simply be redundant.

Instead, I can get rid of that line here

and I can now still access `summaryData` like a property

or like a variable that does not change,

even though under the hood it's a method.

But because of this `get` keyword,

this in the end can be used like a property

by accessing it like this,

but internally it behaves like a method.

So that's another Dart feature which you can use

as an alternative to defining a method.

And it's great for use cases like this

where you have a method that in the end just transforms data

that's stored in another property of the class.

Another example for a method that could be converted

to a getter can be found in the `quiz-question` Dart file.

There we have this `getShuffledAnswers` method,

which in the end also is just such a getter.

It takes another class property, `answers`,

and then derives some data based on that property.

So here we could also rename this to `shuffledAnswers`

to describe what's inside of it, remove the parentheses
and add `get` in front of it.

And now this could be used like a property
whilst internally by Dart

it would still be executed as a method.

Of course here I now have to go to the place
where I called `getShuffledAnswers`

and change this from a method call to a method
that doesn't exist anymore to a property access
like this without parentheses.

So that's the idea behind getters.

Again, using them is optional.

You can stick to methods, but this is a bit shorter
and a feature you will also see being used in many projects.

You will also see it from time to time in this course
and you should therefore know about getters.

Now the last optional Dart feature

I wanna dive in are so-called arrow functions.

And for that I'm back here in the results screen widget
because there I have a great example for a place
where an arrow function could be used.

Now, what is an arrow function?

Well, in your Dart and Flutter projects,
you'll often have functions like this,
this anonymous function here, which takes some input
and then in its function body immediately returns a value.

Especially when using those list methods
like `where` or `map`,

it's quite common that you don't have
a lot of complex code in that function body,
but that you instead only have a single return statement
in there where you then return some derived value.

Now if you have such a case, so a simple anonymous function, which might take some input values and which then immediately returns and performs no other operations inside of the function body.

If you have a case like this, you can change this code to add an arrow consisting of an equal sign and a greater sign so that it looks like an arrow.

And then you can remove those curly braces opening and closing and the return keyword.

And you can not just do that, you have to do that and instead write it like this.

You also need to remove that semicolon at the end.

This then is a so-called arrow function, and it's simply more concise than writing a function with the curly braces and the extra return keyword.

So this feature exists to allow you to write more concise code.

Using it is optional.

There's nothing wrong with using that other function approach, but this simply is a bit shorter and that's why you might want use this approach for functions that only have a return statement in their function body.

72. Module Summary

And that's it for this course section.

In this course section, we built this Flutter learning quiz, where we can answer Flutter related questions.

And after we answer these questions, we get this result screen

where we can see which questions were answered correctly.

And which answers would've been the correct answers for the different questions.

From this results screen, we can then, of course, also restart the quiz, and give it another try where we might do better.

Now, whilst building this app, we again learned way more about Flutter and also Dart. We, of course, used the knowledge we already gained in the previous course section.

For example, the knowledge that this main function in the main .dart file is the main entry point since it's executed automatically by Dart when the app starts.

And that you must then call runApp and pass your widget tree to it.

But in this section, we built way more widgets.

We passed data around between those widgets, including more complex data like functions.

In this section, you learned that in your widgets, you can actually also accept functions as input values, as arguments.

And that you can then use these functions in your widgets, for example, to bind them to buttons.

And that you can therefore also create and then set those functions as values for those arguments that want functions.

You learn that you can pass functions as values, which is a crucial pattern that will be needed quite a lot when building more complex Flutter apps.

You also learned that you can output conditional content with help of the if keyword, for example, which allows you

to set a variable in this case here to different values based on different state values.

And then this variable can be used to output it in the widget tree.

And therefore, since the variable contains different values based on different state values, the UI looks different for those different states.

And that's exactly what conditional content is all about, having different UI states for different conditions.

You did also not just learn about the if statement, which is a great way of setting content conditionally, but you also learned about using ternary expressions, which gives you a more concise, but sometimes also harder to read and understand way of setting content conditionally.

And you even learned that you could manage entire widgets as state values.

Though you also learned that this can have some disadvantages, especially if those widgets then must receive some data, like for example, a pointer to a function.

But you learned about these different approaches.

And you did also not just learn about if statements, but also about for loops, which is another crucial Dart concept.

Actually a feature you find in many programming languages, as it allows you to execute the same piece of code repeatedly until some condition is not met anymore.

And here we use the for loop to go through all the answers chosen by a user, and create some summary data pieces based on these answers.

And these summary data pieces were actually so-called maps.

Another built in value type in Dart,
which allows you to quickly combine different
key value pairs into one object.

The alternative would've been to build a custom class
as a blueprint for a custom object.

And this would've also yielded various benefits,
especially regarding type safety,
because you did see that when using a map,
you have to do a lot of type casting to tell Dart
which type of value is stored
under a specific key in such a map.

But it's nonetheless very important
to know about this map value, since you will also encounter
and use that quite a bit when working with Flutter.

In addition to these Dart features and patterns,
we also learned about some new widgets, like for example,
the single child scroll view widget,
which can be used to make its child widget,
often a column, scrollable.

Which is something we use here on this final results page
to make this box in the middle here scrollable,
whilst still restricting it in size, in height specifically.

You also saw some new configuration items
and styling options in action, like for example,
the shape option for elevated buttons
to add rounded borders to the button.

And in general you, of course, saw many examples
for the usage and configuration of various Flutter widgets.

Last, but not least, in this section,
we also used the class keyword
for something else than a custom widget.

Of course, you will very often create custom widgets, stateless widgets, or stateful widgets.

But in Dart, you can also use the class keyword to build standard classes, standard blueprints for standard objects that don't act as widgets to, for example, build a blueprint for an object that contains the question text and its possible answers. As well as a utility method for shuffling those answers, and returning a new list of shuffled answers.

That is what we used a custom class for in this section. And speaking of shuffling answers, in this section you also learned about various methods that can be called on lists to manipulate or analyze those lists.

For example, the shuffle method, which can be called on a list to change the order of list items.

But also, the add method to add new entries to a list. And the where method to filter a list, to create a new list that only contains the list entries where a certain condition is met.

In addition, you also learned about the map method, which can also be called on a list to generate a new list of transformed list items, which we, for example, used to transform our list of string answers to a list of tapable answer button widgets.

You also learned about this special spread operator here, the three dots operator, which can be used to pull all the list elements out of a list and add them as standalone comma separated items to a surrounding list.

In this case here, used to get

all the transformed answer button widgets

out of that answer button widget list.

And add them as standalone comma separated items

to the list that is passed as a value

to the children argument of the surrounding column.

And therefore, at this point,

you have a solid Flutter foundation.

You learned about many key Dart and Flutter concepts

and patterns, concepts and patterns you will need

for the rest of this course.

And probably for pretty much all Flutter apps

you are about to build.

And I can really only recommend that you also

use this knowledge and build some basic demo apps.

In this course, we're now going to dive even deeper

into Flutter in the next course sections.

We're again going to see all these things and features

we already learned about in action,

hence also practicing those features again.

But we're also going to learn about

many new crucial features.

3_debugging

1. Module Introduction

Now that you have a solid Flutter and Dart foundation, it's important to make sure that you can also debug and fix your apps when things go wrong. Because when writing code, no matter which programming language you're using, and no matter which kind of application you're building, things will eventually go wrong. And as a developer, you therefore must know how you can help yourself and how you can fix things. So in order to give you the tools you need to fix problems and errors you might be facing when building Flutter apps, we'll take a look at how you can read and understand error messages you might be getting when running your apps. We'll also take a look at the Debug Mode you can use for running your Flutter apps and what you can do in that mode. And we'll get started with the Flutter DevTools in this course section.

2. The Starting Project & A Problem

Now for this course section, I took the finished project from the previous section and added one or two problems and bugs to it. And you find this updated project,

this starting project for this section,
therefore, attached to this lecture.
As mentioned, it's essentially the same project as before,
but with some issues.
So if you do have an emulator up and running,
you can run this project without debugging,
as we always did it throughout this course,
and run it on your emulator on your virtual device.
And if you start the app there,
you will see that initially it seems to work as expected,
but eventually you'll face this error here
when you reach the result screen.
And if you then restart the quiz thereafter,
you will notice that if you reach the result screen again,
you face another error,
which now affects the overall screen.
So this app clearly has some issues.
And what can you do now to fix those issues?

3. Understanding Error Messages

Now I wanna start with fixing this
first error here, which I get after going through
all the questions, even if I'm just going
through those questions the first time.
And to fix this error and to get rid of this bug,
I wanna take a closer look at the error messages presented
by a flutter because those indeed very often help you
analyze what's going wrong and help you find
and fix the problem that is causing the error.
Now here you can see that if things do go wrong
and your app throws an error whilst running, you see
that error message right here on the screen.

And by the way, this of course is a feature that only exists during development.

If you built the app for production and distributed it through the app stores, your users will not see those error messages.

There the app will very likely just crash.

But during development, you see the error message right here on the screen and this error message might sometimes already be all you need to find out what's causing this error.

Here, for example, we learned that the error seems to be related to type casting.

And this already might be a good hint because in this application that we have here, there is only one place where we are doing some type casting.

And that's in this summary item where I am casting some types to tell Dart that certain values in my map are of certain types, but the error message that's shown directly inside of the app might not always be enough to find out what's going wrong.

And for that reason, it can be helpful to explore a more complete error output and dive into the error message in greater detail.

And you can do that when using Visual Studio code by opening the panel under view appearance panel, or by using the panel shortcut you see here.

If you open that panel, you should go to the debug console here at the bottom and it's inside of this debug console where you get more details about the error that was thrown.

For example, here you see the error message
and it even tells you which widget was causing this error
message or was related to this error message.

And you also get a link to the file
and even the line of code
that caused the error though if you click on this link here,
you see that it in the end takes you
to the questions summary widget
and there to the line where the summary item is used,
which makes sense because it's the summary item widget
that was causing the error.

But you can get more details about the error
if you scroll up.

And that's in general a good idea.

When you get an error in your flutter app,
you can always open the debug console and scroll up.

You should at some point reach such a stack trace.

So such a long list of involved method calls,
but most of those calls will be internal calls
by flutter related to the code written by the flutter team.

And you can ignore those.

But if you scroll to the top of this stack trace,
you will find more information that's relevant to you.

For example, here we get the error message again,
and sometimes you don't have an error message
here at the bottom.

In that case, you should always scroll up
to see the actual error message.

And in addition to this error message,
you also again learn which widget was related to the error.

And very important, you should get at least one line in
that stack trace here that's highlighted.

In this case the first line,
though it's not always necessarily the first line
and very often this highlighted line is the best entry point
you can use to learn more about the error
or to analyze what's causing this error.

In this case, I for example, see that it's the build method
of the summary item widget that seems
to be causing this error.

And I even see a link to that file here,
including the code line that seems to be causing the error.

And if I click this link now, I am indeed taken
to this summary item widget here
and I'm taken right to the code line in my case here,
line 25, that seems to be causing this error.

Now combined with the error message, the problem here seems
to be that we are trying to cast something to int so
that we're trying to tell Dart
that some value is of type int.

That's what we're doing here with this piece of code,
when the value actually is of type string and dart
therefore complains here at runtime when executing the app
that it can't treat a string
as an integer even if we wanted to.

So the problem here seems to be that the value
that we think is an end is actually a string.

And this is a great starting point
to dive deeper into this error
and into this bug that seems to exist in this app.

What is inside of item data question?

Well, we can go to the results screen, which is
where this map object here is constructed.

And there we can take a look at the value

that has this question key
because it is this question key we're accessing here.
Now if we take a look at that key, we can indeed confirm
that there seems to be a string stored
as a value for that key.
So indeed here the problem simply is
that we are accidentally accessing the wrong key.
This here should be question index
because it's that question index key
that contains an integer as a value.
Now because this map has a value type of object
to accept different value types, DART was not able
to catch this error during development.
It doesn't show any error here in the IDE when using the
wrong key because it simply doesn't store any specific type
information about the values.
Instead, it only knows that any kind
of value is accepted here for those keys.
But of course at runtime, when this code here
really executes, this crashes
because question as it turns out, stores a string
and not an integer.
So here the fix is to use question index instead,
which is the correct key that does store an integer.
And if we then force a reload
and we go through all these questions again, you indeed see
that now the error went away
and we can see those results again.
We also got no error message in the debug console anymore,
and that's how you should treat error messages.
Treat them as starting points to find out what's wrong.
Take a look at the line of code in the code file

that's causing the error
and carefully read the error message.
In this case, you learned that it was related
to type casting and
therefore it made a lot of sense
to take a closer look at this line of code.
Of course, you then still in this case, had
to find the error on your own.
But the error message of course helped a lot
with getting started with finding that error.

4. Debugging Apps & Using "Debug Mode"

Now understanding error messages
is a crucial skill,
but it's not the only thing you should be able
to do as a developer.
And to show you another helpful feature that you can use
for debugging your app
and for finding error sources,
I'll move on and restart this quiz
because as shown earlier, if I go through it again,
I now get an error again,
suddenly a different error this time affecting the entire
screen and also different error message but still an error.
So what's the problem here?
Well of course you could
and typically should also again take a look
at the error message here.
Also in the debug console this time there is no error
message at the bottom,
but as mentioned, if you scroll to the top
of this stack trace, you find an error message here.

And here we seem to have a problem with using an index in a list most likely where we try to use an index six in this case, in a list where we only have indexes zero to five. So we seem to access a list item that simply doesn't exist. And of course, again, we learn which part of our code is related to this error.

We see that it seems to be related to an array, which in the end is simply a list. But then we also learned that it seems to be stemming from the build method in the questions screen state class.

And again, we can click this link to be taken to that code file and to that line of code to find out where this error is originating from. But of course we could now think about this app again and manually go through all the code again.

But instead I want to show you another feature here. I wanna show you how you can run your apps in debug mode and how you can use certain debugging features.

Now to use that debug mode, you first of all should stop the running app and then restart it, but this time by starting it with debugging. So by using the start debugging option.

Alternatively you can go to the main dart file and there click on debug.

This starts the app not just in development mode but in debugging mode, which unlocks some extra features that make the app a bit slower, but help you as a developer with analyzing the app whilst it runs.

So it's not the mode you should use as a default

for running the app since it will slow the app down,
but it is a great mode for getting access
to some extra features that can help
with tracking down errors.

So here the app started again, and you can use it as before,
but now with debugging mode started,
you can do one special thing.

You can add break points in your code
and you do that by going to one of your code files,
here, for example, to the code file
and to the line of code
that was mentioned in the error message before.

And if you then hover to the left of these line numbers,
you see this red dot here.

If you click on that red dot,
you add a so-called break point.

And what does this break point do?

Well, let me answer a question and show you.

As soon as this line of code
where I added my breakpoint executes, app execution stops
and I moved back here to my editor
and I can now analyze my code
and the values that are currently being used in this code.

So the values that are currently stored in properties
and variables, for example at runtime.

And app execution is paused, as you can see,
it's frozen basically until I continue
by clicking this play button here.

And this all works despite the start code being compiled
to machine code fluttered us all the heavy lifting
of linking the app execution on my device,
which happens in machine code back

to my original source code.

And that's this extra work that's being done by flutter behind the scenes in debug mode.

Now here I can for example, hover over values, for example, the current question index property here to see the value that's currently stored inside of that property.

And I cannot just hover over current question index, but for example also over questions.

And I would see that this is a list with six items and I even see the list indexes here and I can even expand the individual items to see which values are stored inside of these items.

And these features can of course give you very valuable insights into your app at specific points of time whilst the app is running.

Now here in this case I'll continue because currently the app is still working.

But of course now every time I am answering a question, this gets paused and I can analyze the current values again.

You can also right click on a property or variable and add it to watch so that it is added here to the left in this watch window.

And now you will always see this variable or property with its latest value without even hovering over it.

So if I now continue and answer another question, I see the updated value here without even hovering over current question index.

So this can be very convenient if you know that a specific value should be tracked and you want to quickly see what's inside of that value.

Here for example, I now see that I have an index of five and I do have six questions with the last question, having an index of five.

So accessing this question should still work, but if the current question index would thereafter increase to six, we would probably get an error.

By the way here, you don't just see the watch window on the left, but you also have some generally available variables here.

For example, this refers to the entire question screen state object, and you could dive deeper into some internal properties here.

Many of those properties being managed by flutter and context is this extra object which I mentioned before, which we will use in the next section and which we can ignore for now.

You'll also see the current call stack here, which simply allows you to find out which methods have all been called under the hood up to this point where this breakpoint was reached to get the app to where it is right now.

But all that doesn't help us right now.

So here I'll just continue and I'll now go to the result screen and restart the quiz.

Now it pauses again and now we can indeed see that the current question index is reset to zero here.

So this seems to work correctly.

So indeed the error seems not to be related to the current question index being too high, at least not right now.

Here it was reset as expected.

So we can continue a couple of times.
And since I know that the error won't occur
until I reached the result screen again,
I'll also disable the breakpoint again
by clicking on it again and by resuming
and then answering all those questions except
for the last one
because I know that thereafter I'll get my error.
Here I'll reintroduce my breakpoint
and then click and answer.
And now it pauses again.
And here I now see
that suddenly current question index is six,
even though it was reset before.
And this is of course the reason
for this error we're getting
because we only have six questions
where the last question has an index of five.
So trying to access a question
with index six will simply fail
because there is no such item in that list
and that's why we're getting this error,
which we get thereafter.
And if an error occurs as this case here,
it will actually automatically pause the app even if there
wouldn't be a breakpoint at the point of time
where the error occurs.
But of course, thus far we haven't learned anything new
or of course not too many new things.
We just know that we got an index that is too high,
but we need to find out why it is too high.
And for that, we should take a look at the rest

of this class here.

There we can see that we're initializing the index here
and we're incrementing it whenever a question is answered.

Indeed, there is no code in this entire class here
that would ever reset the question index back to zero.

We only do that when this class is instantiated
for the first time.

So the problem here indeed seems to be
that we never leave this class
that we stay in this class
and that normally should not be the case.

If we switch back here to the explorer
and then take a look at the quiz dart file,
we of course can see that in there
we normally should leave the questions screen
and navigate to the results screen as soon as we have
as many answers as we have questions.

So it seems to be this navigation that's broken here
because the problem seems to be
that we're staying on this questions screen for too long.
We're staying on this screen
until we reach a current question index that is too high,
that's the problem.

And so therefore, the actual problem seems to be
that we never leave this questions screen.

And since this seems to be the problem,
it probably makes most sense to restart this app
and to again go back through that quiz once
and then another time almost to the point
where we get this error.

But now here before we get this error,
I want to add some break points here in quiz dart.

And I wanna add a break point here in choose answer
because that code is executed
whenever we choose an answer.

Here in questions screen,
we are calling on select answer,
which in the end is a function, which in the end is
that function defined here in quiz dart.

And I wanna take a look at choose answer
to find out if there's something wrong with this condition,
which should normally take us away
from the questions screen.

In addition I'll add a breakpoint in the build method here
to find out which widget we're actually displaying
because it seems to be the questions screen in a situation
where the questions screen shouldn't be visible anymore.
And with that, I'll answer this last question.

And now it pauses here in choose answer.

And here we can of course also take a look at the various
values we have here, for example,
the answer that was selected and then add selected answers.

And here we can see a problem.

We got a super long list of selected answers here.

And keep in mind that we only had six questions,
but here we have 11 answers already
and we're about to add another answer.

Now, besides resuming, we can also just continue
with code execution step by step
by using this second button.

So now we go through the different parts of this instruction
and then to the next line of code.

And this can be very useful for walking
through your code at runtime step by step.

Here I used this feature to finish adding this answer to selected answers so that indeed thereafter I can confirm that we have 12 answers in there.

And since we have only six questions, the problem indeed seems to be that we never reset the selected answers list.

We never set it back to an empty list after answering all questions for the first time.

And for that reason, this condition here will not be met because here I'm comparing the length of selected answers, which is 12 in this case, to the length of questions, which is six.

Of course this is not equal.

And therefore if I step over, we can see that we don't make it into this if block the code in there was never highlighted, it was never executed.

And instead, if I continue here, we see that the next main step thereafter basically is this error that's being thrown.

So we can also see that we never reached this break point here in the build method.

This build method never reexecuted, so the build method of the quiz state class, and this of course is a clear indicator that active screen, this state was never set because if it would've been set with set state, this build method should have executed before any error occurred.

But that never happened.

And we therefore now know with certainty that we never navigated away from the questions screen

that we indeed still are on that screen
and that the problem seems to be
that this condition here was never met
because selected answers wasn't cleared.
And indeed, that here is the problem.
Here in restart quiz,
we're not setting selected answers back to an empty list
when we actually should be doing that.
Because here we're going back to the questions screen.
We are restarting the quiz,
but restarting the quiz also involves resetting the answers.
If we're not resetting the answers,
this condition here is never met
and we never make it to the results screen.
We therefore stay on the questions screen
and there eventually we reach an index that is too high.
That's the problem here.
And using this debugging tool can make it a bit easier
to find this problem and to fix this error.
So here for example, I now know
that I could try adding selected answers
equal and empty list here in the set state function call
of the restart quiz method in the quiz state class.
And with that added, we can try restarting the app with
or without debugging.
And now hopefully we should not be getting
this error anymore
because now with help of the debugging tool,
we were able to track down the error
or at least the potential error
and we hopefully were able to come up with a fix.
So let's see if this works now.

If I go through this quiz twice again,
first time works as before, but what about the second time?
This now also works
because indeed it looks like this was the problem here.
And as you saw, the debugging mode can help you analyze
your code at runtime and it can therefore help you find
and fix such errors.

5. Working with the Flutter DevTools

So error messages
and debug mode can be very helpful
when it comes to finding and fixing errors.
But you also got one last tool at your disposal,
which I also wanna show you,
and that would be the Flutter Developer Tools.
Now, when using Visual Studio Code,
you might sometimes be asked when starting your app,
whether you want to open those DevTools,
and of course, you can open them by using this pop-up,
but you can always open them
by going to View, Command Palette,
or by using this shortcut here in Visual Studio Code.
And then here you can type Flutter DevTools
and choose this Open DevTools option.
And here, you can then actually open different parts
or different pages of those Developer Tools.
But I personally recommend using this option here,
Open DevTools in Web Browser.
This opens the Developer Tools in a new browser window,
and this window
and these tools here are connected to your running app.
And you got various pages here

that help you analyze this app whilst it's running.

For example, you got the Logging page
where you see various log messages
and also potential error messages you might be getting.

You've got the Network page that can help you
with debugging network requests
that are being sent by the app.

And at the moment,
we're not sending any network requests,
but we'll do that later in the course.

You can analyze your app size.

You can analyze the app performance.

Though, if that is what you want to do,
as mentioned here,

you should use profile mode
by starting the app in profile mode
by clicking this Profile button here
above the main function.

Because in debug
and also in the regular development run mode,
there are too many helper tools going on behind the scenes
by Flutter to help you with development,
but those tools all slow down the app.

And in profile mode, there is less of that,
and therefore, it's this profile mode
that could help you improve app performance
and analyze how performant your app is.

You also get a Debugger here
in case you don't want to use
the debugging features built into Visual Studio Code,
which I just showed you.

But for this section here,

I wanna focus on this Flutter Inspector
because this Inspector can help you with analyzing
and optimizing your layouts
and inspecting your widgets.

For that, you will see
your currently rendered widget tree here,
and that's basically your UI,
which you see here on the screen on the left,
translated to a nested widget tree,
where you can see how those different widgets
are related to each other.

And this can be very useful
for understanding what's going on on your user interface
and which widgets are actually involved
in that user interface.

But, of course, you can also select those different widgets
and learn a bit more about them.

You always get a Layout Explorer here on the right,
where you can see the width
and the height of a selected widget
and its relation to its parent widget.

And if you hover over a widget here in this widget tree,
you also get a window
that helps you dive into the actual widget object
that's being rendered behind the scenes by Flutter.

And you can, for example, explore its state
or any other properties that are set on that object.

And this is, of course, not just available for MaterialApp,
but for all your widgets.

Now when selecting a column or a row here,
you even have an interactive preview here on the right,
or an interactive Layout Explorer, to be precise,

on the right,
where you can adjust the main
and cross axis alignment directly at runtime
from inside those Flutter DevTools.
You can, of course, always do that from inside your code
by editing `crossAxisAlignment`
and `mainAxisAlignment` in here.
And if you then save your code changes,
the UI would update automatically as well,
as you saw throughout this course already.
But you can do it in a even more convenient way
by using these dropdowns here
to quickly try out different settings
and see how your user interface changes
if you change those settings.
For example, here you can see that the identifier
of the row I selected here moves around
if I switch between start, end, and center
for my cross axis alignment for this row.
And you can do this for all your column
and row widgets that you have here.
And this can be very useful
for quickly trying out different options
and finding out why certain elements
are positioned the way they are positioned.
In addition, even for non-column and row widgets,
you get this Layout Explorer here on the right,
which helps you understand which width
and height the selected item has
and how it's related to its parent widget.
And you have this Widget Details Tree
into which you can dive,

where you can also again analyze the different properties and settings that are set on that widget, for example, for this Text widget, the text that's added to it, the color that was chosen, the size that was automatically derived, and many other things as well.

And these can all be very useful features for analyzing your UI.

In addition,

if you know that you wanna analyze a specific widget but you don't wanna search for it manually in this typically quite long widget tree here, you can use this button to select the widget from inside the screen and to also see the widget that is currently selected.

So here, if I know

that it's actually this chosen answer here which I wanna inspect,

I can click on it after clicking on this button, and it will then be selected here in this widget tree.

And I can then again use the Details Tree and the Layout Explorer.

If I then wanna select the another item,

I just have to click on this magnifying glass here at the bottom corner, and then I can select the another item.

So that's all very useful.

I can always disable this mode, of course, if I'm done for the day.

And in addition to that,

I also got some other features here,

where I can, for example,
see some guidelines to understand
why things are positioned the way they are positioned
and how they are related to each other,
for example, why a certain item is centered
or anything like that.

I can show text baselines to make sure
that all my text is aligned the way I want it to be aligned.

And if I restart this app here,
if I'm on a screen that contains an image,
I can also use this last button here
to highlight images that are basically too big.

In this case, nothing changes
because this image here isn't too big.

But in other apps,
you might indeed get some hint here
that an image is unnecessarily big
because you're resizing it
such that it's way smaller than the original image
in the file,

and in such cases,
you might want to decrease the image in the file as well
so that you're not unnecessarily loading
a too big image into memory.

Well, and that's it for the DevTools.

Definitely feel free to play around with them.

Combined with the debug mode I showed you
and the error messages that are shown
by Flutter if things go wrong,
you now do have a solid set of tools
that should allow you to find, understand, and fix errors,
and it should allow you to build exactly the apps

and the app user interfaces that you want to build.

4_expense-tracker_interactive-widget-theming

1. Module Introduction

So now that we spend quite a bit
of time diving into those crucial Dart and Flutter features,
it's time to dive deeper into Flutter.

And in this course section, we are going to explore
how we can add more interactivity to our Flutter apps,
how we can handle more than simple button clicks,
and also how we can theme our Flutter apps
to apply some global styling to our Flutter apps.

We'll explore how we can show modal overlays
and what exactly that is,
how we can present dialogues to users.
And we'll also learn about additional interactivity elements
we might want to use in our apps.

We'll also learn how we can handle basic user input,
how we can validate user input to make sure
that we're working with valid data.

And as mentioned before, we'll dive into themeing
and how we can easily style our entire app
from one single place.

Now to explore all these topics and more,
we'll build another brand new app,
an expense tracker app that allows users
to register their expenses,
and that then also allows users
to see their expenses both in a list and in a chart.
We're going to build this application here
from the ground up with all the features you can see here.

And along the way, whilst building this app,
we will, of course, practice what you already learned
and dive into all these exciting new features.

2. Starting Setup & Repetition Time!

So let's dive right in.

And for that you of course should create another new flutter
project with the flutter create command.

Here, I created a project called EXPENSE_TRACKER
with flutter create expense tracker.

And then attached

to this lecture you find a main.dart file,
which you should use

to replace your main.dart file in your project with
or simply copy the content of the attached main.dart file
and replace your main.dart file's content with it.

And also make sure you create an expenses.dart file,
which at the moment will be empty

because that is the file

where I want to create my main user interface
or where I want to get started with creating that interface.

And to get started, we of course,
first of all should take another look at that interface
and at the core features we have there.

And if you take a look at the finished app, you see
that there are a couple of things on that user interface.

In the end we have that chart,
and then below that chart we have that list of expenses.

But of course we also have this bar,
this toolbar at the top you could say,
which has this icon button, which we can tap
to open this modal overlay here, which then allows us

to insert a new expense.

So that's also something we'll have to add.

Therefore, these expenses widget here in the end should give us these three main areas here on that starting screen, that toolbar or app bar as it's called in Flutter's world at the top.

Then below that, the chart,

and then below that the list of expenses.

And we'll take care about the modal overlay and so on later.

For that in `expenses.dart` here,

I'll create a new widget class and I'll name it `expenses`.

And this should actually be a `Stateful Widget` which we built here because of course at some point we want to start managing expenses.

We want to start adding expenses, for example.

And that then will of course require a UI update because every time we add a new expense, we want to, well, update the UI to show that new expense in the list and to also update the chart.

And since we have that list and chart, and both are part of that main expenses screen, which should be this expenses widget here,

it's this expenses widget

that should manage the data in a `Stateful` way because you learned that we need a `StatefulWidget` if we have data which we want to change, and where those changes should then update the UI.

Therefore, here we extend `StatefulWidget` as we did it multiple times before in this course.

And you also must add this import to `material.dart` of course.

And in that StatefulWidget,
we have this createState method, which we should add.
And you also learned, of course,
that StatefulWidget in the end are widgets
that are created with help of two classes.
The first class is the widget class itself,
and the second class is a state class,
which is typically named underscore,
and then your widget name in this case ExpensesState.
And here we want to extend the built in State class,
which is provided by Flutter, which is a generic class
where we basically have to tell Flutter which kind of state
or to which kind of widget this state will be connected.
And here that's the expenses widget.
And in this expenses state class here,
you then must add the build method like this.
And in the build method we'll then
of course return our widget tree.
Now in the createState method in the widget class,
we then should return a new instance
of our ExpensesState created like this.
And we can also change this state here
or this type here to be State of Expenses,
to be a bit more specific, which kind
of State we're creating here.
With that, we of course also must add our constructor
function here to the widget class,
or we should add it at least.
And it can be a const constructor.
And at the moment here, I just want to forward
that named key argument to the parent class.
So with that, we got the basic widget setup

for this expenses widget.

And before, we'll now start working on that widget tree that we want to return.

We can therefore go back to the main dart file and there import from our own package from our own project here, without the typo.

So, import and here my project is named expense_tracker, and I want import from expenses dart, and then set my home widget here to my expenses widget, which I can now use.

And since there we allow for a const constructor, and MaterialApp also supports const constructors, we can add const in front of MaterialApp.

With that, we got that all set up.

And now it's time to output something on the screen.

For this year, of course, we have to return a widget in our build method.

And for the widget that should be returned

I'll pick this Scaffold widget here,

which is another widget we used quite a bit before because this widget does some general base style setup work.

And it, for example, also sets the background colour of the app to white.

So here I want to return Scaffold and then add my body here and set the body of this page here, of this screen, so to say, to a Column widget.

And I want to use a Column widget here because in the end I want to have this chart above this list of expenses.

So I want to have two widgets above each other.

And that basically always means that you want to use a column because that allows you

to add multiple widgets above each other.

So here we then set the child widgets of that column, so the widgets that should be above each other.

And here we'll soon have the chart and then the list.

For the moment, I'll just add some Text placeholders, which will be replaced later.

So here I'll have the chart and this can be const, or actually we can even add const in front of this entire list.

And then I have another Text widget where I will have my Expenses list later on.

As mentioned, these are just placeholders, of course.

Now, if we run this application here again on an Android emulator, in my case, we should have this white background and also this text here squeezed into the top left border. But of course, we'll change the styling soon.

3. Adding an Expense Data Model with a Unique ID & Exploring Initializer Lists

So now we get started with this basic user interface here.

How should we now continue?

Well, there are of course different ways of continuing, and of course we will need to build this chart and this expense list widget ultimately.

But I wanna start by creating a model as a first step in a separate models folder that I just created here in this lib folder.

Of course you don't have to create this models folder, but I want to organize my files and kind of group them by functionality or feature.

Inside of this models folder, we will create a dart file, which I'll name expense dot dart, where I simply want

to describe which data structure
and expense in this app should have.
So which kind of structure and expense should have.
And for that in this expense dart file,
we can create a class which now won't be a widget.
So where we won't extend state full or state less widget.
And it's a class which will name expense.
And of course we have created a data model
before in this course already here in this app.
I'll do it again because we'll work a lot
with such expenses.
And a single expense will be made up
of different data points, which I simply want
to group together in such a custom class,
in such a custom object blueprint.
So which kind of data should be grouped
into a single expense?
Well, I definitely want to have the title there.
So add a property, a final property
of type string, which I'll name, title.
I also definitely want to have the amount,
and that should be of type double, which is a number
with decimal places.
So that we also support amounts like one dot
99 for example.
And besides title
and amount, I also want to have a unique ID per expense,
which then allows us to identify that expense later on,
which is some information that we, for example,
need to delete expenses.
So therefore I'll add an ID property,
and this will also be of type string.

You could also use an integer ID
to have IDs like one, two, three, and so on.
But here I'll generate a string id,
some unique strings so
that every expense has its own unique id.
And we'll soon add some code that takes care
of generating such an ID automatically.
Now I also want to have a date attached to every expense.
And for this we can use the built in date time data type,
which is a data type built right into dart.
So we don't have to import anything,
we can use it like this.
And date time is simply a special data type that allows us
to store date information in a single value.
And with that here I added four properties that
for this application, which we're building here,
should describe a single expense.
Now of course, we also need a constructor function
that can be called
to well, build such an expense object
based on this blueprint.
And therefore here I want to accept a couple of arguments,
a couple of parameters,
and we could use positional parameters here.
But here, since I'll need a couple of parameters,
I'll actually go for named parameters
because if you expect more than two
or three parameters in a function
or in a constructive function like here,
I personally find it easier to call
that constructive function
if you can target the different parameters by name instead

of having to remember their position
and passing data in the right order to that function.
Using named parameters can simply be a bit easier here.
Therefore, here I wanna accept a named title parameter
to assign a value to title.
And this should be required with help
of the required keyword here.
The same is the case for the amount.
I wanna accept this as well and it should also be required.
Then I also wanna require the date parameter.
And for the ID here, I actually don't wanna require this
as a parameter.
Instead, I want to build such a unique ID dynamically
whenever a new expense object is created.
And to do that, I'll use a third party package,
the UUID package,
which you can install into your flutter project
with this command here like this,
so that we can use this package to generate unique IDs
because that's what this package is all about.
So with this installed here in this expense dot dart file,
we can import from this UUID package
and they are from the UUID DART file.
And this then gives us an object
or a class to be precise, which we can instantiate
to get an object,
the UUID class,
which we can call like this,
to create an object based on it, which I'll store
in a final variable called UUID.
And I do this outside of my class here
because it doesn't belong to this class necessarily.

Instead it's just a utility object,
which we can now use anywhere in this file
to generate unique IDs.
And I want to generate a unique ID for this expense class
whenever it is instantiated.
And for this we can add a so-called initializer list
after the closing parenthesis
of this constructor function here by adding a colon.
And after this colon, you can also initialize the properties
of this class to which this constructor function belongs.
So therefore, here we can then, for example,
target the ID property here
and assign an initial value
that will be assigned when this object is created.
So when this constructor function is called,
and here I want to use this UUID utility object.
And on that object there is a V four method, along
with a couple of other methods,
but it's the V four method, which I want to use here,
which generates a unique string id.
So this then generates a unique ID
and assigns it as an initial value to the ID property
whenever this expense class is initialized.
This initializer list feature here is a dart feature.
UUID is this third party package which we added.
And here we can now also add const since UUID supports this
const constructor.
And since we can add it here,
we can actually also add it here
to turn this entire variable into a const value,
which we can't reassign
and which we create such

that the value can be stored in memory
and could be reused
by dart if we would use it somewhere else in our code
as well.

4. Introducing Enums

So now we got started
creating this Expense class and we're almost done.
But there's actually one piece of information missing
that should also belong to Expense objects,
and that's the category to which an expense belongs.
Because in this finished app, which we're building,
every expense has a category attached to it,
which is important for that chart that will be built,
because that chart groups expenses by category.
So therefore here I also wanna store some category
information in my expense class.

And you could now of course say
that you wanna to have a category property,
which is of type String.
This would allow us to add categories as string values
to these expenses as we create them.

But this could be quite error prone,
because if you, for example,
wanna support the category leisure like this,
we could of course create such a Expense object
by calling Expense and passing leisure
like this as a category value.

But it could happen
that we accidentally introduce a typo here,
and we wouldn't get any warning or error when doing so,
because any kind of string value is accepted.

So even though that might not be one of the categories we wanna support to this app, we wouldn't get an error, because it is a valid string. It just wouldn't show up in our chart, because there we might not look for this category because it contains a typo.

So for that reason, using a string here is not optimal. It's too flexible, it allows too many values. Instead, we wanna create a fixed set of allowed values so that we have to use one of those values when creating a new expense.

And for that, we can create a new value that contains all these allowed values with another special Dart feature, the enum feature, the enum keyword. enum is a keyword that allows us to create a custom type. You could say that could be named Category here, which simply is a combination of predefined allowed values. You define those allowed values between opening and closing curly braces. So this simply is the enum syntax in Dart. And then here, between those opening and closing curly braces, you list your enum values.

For example, here, for the Expense categories that we wanna support in this app here, I wanna have food, travel, leisure, and work. And of course you could add more though. These will be the categories I will later output in my chart.

So with that, we're simply saying that in this project, in this Dart project, in this app here,

we wanna have a custom type category
that supports these possible values.
Please note that these values are not wrapped in quotes.
So we don't have string values here technically,
but nonetheless,
this entire syntax here is recognized by Dart
and treats these values kind of like string values.
You will see how we can access them
as text later on in this section.
But now with that, we can use Category
as a custom type here for this category value
in our Expense class.
And now of course, we also wanna accept categories
as input values.
And for this here, I will also accept a category parameter,
a named category parameter, which is required.
And that's now the finished Expense model for the moment.
We'll fine tune and change it a little bit later,
but that's the model for the moment.
And with that model defined, we can now switch back
to the UI and make sure that we use that model there.
And for example, output a list of expenses.

5. Creating Dummy Data

So let's now use this expense data model
to create some dummy expenses here in our expenses widget,
or the expenses state class, to be precise,
so that we can then output these dummy widgets
here in this list before we then later will, of course,
also add the logic to create expenses ourselves.
But we'll start with some dummy expenses
so that we have something to output.

And for that, in this expenses state class here,
I'll add a new final list of expenses,
so of using this custom expense class as a type here,
because as you learned before,
whenever you create a class,
you automatically also get a type of the same name
which simply contains information regarding the shape
of the objects that will be created with help
of your class later.

And here, I'll name this variable,
this property in this expenses state class
which I create here registeredExpenses,
and I'll set it equal to a list with square brackets.

And now here I simply wanna create a couple
of dummy expenses by using our expense class
and instantiating objects with help of this class.

And here we now have to pass in
these different pieces of information like, for example,
the title of this expense.

And here I'll give this first dummy expense
a title of Flutter Course
and add an amount of, let's say, 19.99
and set the date to today.

I'll keep it simple and simply use today as a date.

And you can create new date objects
because keep in mind that date wants DateTime values,
so date objects in the end
with help of this DateTime type which we used before,
which is actually not just a type,
but also a class built into dart, as you can tell.

And you can use DateTime to create new dates
by passing in various pieces of information here

to the constructor function

or by simply using the special now constructor function

by using `DateTime.now`.

And this simply uses `today`

and `now` as a date time,

so as the date and time that will be stored

for this expense.

So with that, I'm passing in all that information.

Of course, we now also need to set the category, though,

and this can now be set with help of our enum,

which is available here since I am already importing

from this `expense.dart` file.

My IDE, my code editor, automatically added that import

before when I first referenced my expense class type here,

but you can also add it manually.

But with that file imported,

we also have access to the enum,

since that is defined in the same file,

and we can use that here for creating our expense

by referring to that enum

and then using the dot notation

to access the different predefined values

that make up this enum.

And here we could, for example,

assign a category of work to this first expense

and then copy this expense

and quickly create a second dummy expense here,

which could be named `cinema`.

And maybe here we paid `15.69`.

I'll also use `today` as a date to keep things simple.

But the category here would, of course, be `leisure`.

And with that we have two dummy expenses which we can use.

Now, right now I'm getting a hint here
that I'm not using registered expenses,
but that will change because the next step now is to output
those expenses in a list,
and that's what we'll do in the next lecture.

6. Efficiently Rendering Long Lists with ListView

So let's now output these expenses in a list.
And for that, I'll replace this Text widget here,
which acted as a placeholder, with an actual list.
But I'll not add this list here in this `_ExpensesState`,
instead, I'll create a separate widget
to keep this widget or this state,
which belongs to the Expenses widget, lean,
and to especially keep its build function lean and simple.
Because that is considered a good practice,
keeping your build functions lean
and not putting too much logic into them.
That's typically what you want to strive for.
So therefore, here, I'll add a new file,
`expenses_list.dart` sounds like a fitting file name.
Because in there, I'll add a new class,
the `ExpensesList` class,
which now will be used for creating a widget.
And therefore, this must extend
one of Flutter's widget base classes.
And here a `StatelessWidget` is all we need
because we won't manage any internal data in that list.
Instead, we'll just output the list of expenses
that is defined here.
We don't manage that list, we just output it.
And therefore, this will be a `StatelessWidget`.

And of course, it needs this build method.

And in this build method,

I then want to return my list in the end.

Now, before we create that list,

we, of course, must make sure

that the ExpensesList widget class here

accepts this list of expenses as input.

For that, we need to add the constructor function,

which here can be a const constructor function.

And of course, we want to forward that key as always.

But then I'll accept more named parameters

because I want to have some properties in here.

To be precise, I'll start with one property,

we'll add another property later in this section.

But this one property, which I'll add here,

is my list of expenses.

So here I'll use the built-in List type.

And it's a generic type, as you already know,

and it is basically a list full of expenses.

Hence, I'll add my custom Expense class.

And by using this suggested option here,

my editor also automatically adds this import

to the expense.dart file.

So here I'm using this Expense type here

to combine it with the List type.

And then here this property can be named expenses.

And I now also have this required named expenses parameter,

which is used to assign a value to this expenses property.

And now we can use this expenses property here

in the build method to output this list of expenses.

Now, thus far in the course,

whenever we had multiple items

that should be displayed above each other,
we used the Column widget.

And this is a great widget
for outputting widgets above each other.

You can use it for a list content too, therefore.

But if you have lists of unclear length
that could potentially get quite long,
as it is the case here, for this expenses list,
users could add 100 or 1,000 expenses
when the app is up and running.

So when you have a list of unknown length
that will potentially get very long,
Column is not ideal.

Because with Column, all the widgets that you add to it
will be created behind the scenes by Flutter
whenever this widget that outputs this Column
becomes active.

So that means if you have a list with 1,000 items,
all these items will be created behind the scenes by Flutter
at the point of time where this list widget
is displayed on the screen.

The problem with that is, of course,
that only a small amount of those expenses
is actually needed because only a small amount
of the expense list items that will display here
will be visible on the screen.

We will have a scrollable list, of course,
and most of those items, therefore,
won't be visible initially.

Hence, creating all these invisible items
behind the scenes at the beginning
is really redundant and costs a lot of performance.

That's why you should not use Column
for lists where you don't know the length,
but where you might have a lot of items at the end.

Instead, Flutter gives you a different widget
that should be used for such situations,
and that's the ListView widget.

Now ListView can also be created like this,
and you can pass children to it.

If you do that, you get a scrollable list,
which still creates all items immediately
when this list is displayed on the screen.

So the only thing we gain here
is that this is automatically scrollable.

But it's still not ideal for this scenario here
where we might have many items
that won't be visible initially.

Instead, here we should not use ListView like this,
but use a special constructor that's offered by it,
the builder constructor function.

This is a special constructor function
for this ListView widget,
which in the end tells Flutter
that it should create a scrollable list.

Lists are always scrollable lists by default.

But that it should build, create,
those list items only when they are visible
or about to become visible,
not if they're not visible.

For that, builder takes an itemBuilder parameter,
which is of type Function.

So which wants a function as a value.

And of course, you learned before in the course already

that you can pass functions as values for parameters.

So that's what we need to do here.

And it wants a function that will get

two input values itself.

Those will be provided automatically by Flutter.

And it should then return a widget.

So this function, which we pass as a value to itemBuilder,

should return a widget,

the list item that should be created when needed.

So for this here, I'll create an anonymous function.

You could also create a separate method here,

and then just point at it.

But here I'll go for an anonymous function.

And I'll get a context object and an index value

automatically provided by Flutter.

That's what you can see here.

And I'll get back to this index value a little bit later.

And then we must return the widget.

And for this, you could define your function like this,

and then return your widget here.

Or you use this super short function form

by using this arrow, where you then immediately

define the value that should be returned.

And I'll do that here

since all I want to do in this function

is return a value.

And the widget that should be returned here,

in this case here for a start,

will simply be a Text widget,

though we'll change this later.

And here, in this text widget,

I want to output the title of the expense,

which we are displaying here.

Now, for that, of course,

we, first of all, need to tell Flutter

which items it should build with this ListView,

because we're not doing this right now.

To tell Flutter which items should be included,

we can pass another parameter to this builder method,

to this builder constructor function.

itemBuilder is required,

but we can also pass more arguments.

As you can see, a long list of parameters

that can be set, actually.

Now, these different parameters can be used

to configure the behavior of ListView,

and definitely feel free to browse through them,

also with help of the official documentation.

But here it's the itemCount parameter that should be set.

Because itemCount, in the end,

defines how many list items will be rendered eventually.

And for that, of course, I want to pass

the length of this ExpensesList as a value.

So expenses.length.

Because this built-in length property can be used

to get the number of items in this expenses list

that we get as an input here in this widget.

And that number of items is exactly what we need

as a value for itemCount,

so that Flutter knows how many items

should be displayed here.

And that's now where this index becomes important.

Flutter calls this function here,

which we pass as a value to itemBuilder,

as often as defined here by itemCount in the end.

So if itemCount is two,

because we have two items in our list,

this function here will be called twice.

And for every function call,

index will be a different number.

It will be zero for the first item

because list indexes in Dart,

as in many other programming languages,

start at zero, as you already learned.

And it will be one for the second item, therefore.

So therefore, here we can get the title of the expense

for which a list item is being created here

by reaching out to expenses.

So to this expenses property.

Then using the square bracket notation,

since expenses is a list,

to access a specific expense by index.

And then, of course,

since we want to get the title of that expense,

we use the dot notation on that single expense

to which we now got access to access the title.

And with that, we're outputting

this expense title in a Text widget here

with help of this ListView widget provided by Flutter

by using this builder constructor function

to make sure that all the items

are only created when they're needed.

And if you have a long list of expenses,

this can significantly improve

the performance of your application.

7. Using Lists Inside Of Lists

So, with that, we got a first version

of this ExpensesList widget,

and back in expenses.dart,

we can now use this custom widget.

So here, below this chart placeholder,

I'll use ExpensesList.

Of course, you also must add a import

to your expenses_list.dart file here

in order to use this widget here

in the expenses.dart file.

And now we must pass a value

for this named expenses parameter,

and here that's my registeredExpenses list.

That's the list of expenses

that should be displayed on the screen.

Hence we can refer to it like this here.

Now here I have to remove the const keyword

and add it just in front of text

because even though ExpensesList supports

the const keyword in theory,

we can't use const in front of this widget

if we're referring to List property here,

which could be changed later on.

It could be changed despite being final

because we will still be able to add new values

to this list which is one single object in memory.

We just wouldn't be able to assign a new value

to registeredExpenses.

But that's also something I'll get back to

later in the course.

With that, if we save this, we should now see our list here

except for that we don't,
and the reason for us not seeing the list
is that expenses displays a column,
and in that column we now got a list.
So, in the end, another kind of column inside of a column.
If you have combinations like this,
you will always run into problems with Flutter
because it then doesn't know how to size
or how to restrict the inner column,
and you can then solve this problem
by wrapping this inner column or column-like widget
with the Expanded widget
and set the child of Expanded to this inner column
or ListView in this case.
And, with that, we now see our list here.
And here's the text widget above it
which is this chart placeholder.
Of course, again, styling is all not final,
but it's a start,
and we're outputting this dummy list now,
and we're actually doing so in a way
that's optimized for performance.

8. Creating a Custom List Item with the Card & Spacer Widgets

Now that we got our expenses list,
it's time to work on these list items again.
Because of course we got this text here,
but the goal later is to not just output some text,
not just output the titles of those expenses,
but instead to output all the expense
information that's relevant.
For that, I'll create another new file

because I want to create another new widget
which then will be used for these expense items.
Again, to keep the build method
of this expenses list widget relatively lean,
since you will see that the expense item widget,
which I'm about to create,
will actually combine many other widgets again.
So for that, I'll add a new file.
And since this will be a file with a widget
that's closely related to this expenses list,
I'll organize my code such that I add a widgets folder.
And in that widgets folder,
I'll add an expenses list folder.
And I'll then move expenses list dart
into this expenses list folder.
And in that folder, also create an expense underscore item
dart file, which will then be that file
that contains the class, the widget class that will be used
for outputting the individual list items in that list.
And I'll also move the expenses dart file
into the widgets folder, not into expenses list,
but into the widgets folder
so that in the end of my project,
I have the main.dart file,
and then I have the models folder with expense dart
and the widgets folder with expenses dart
and expenses list,
this expenses list folder with its two widgets.
And organizing the code like this
is of course totally optional,
but it can help with ensuring
that your project stays manageable.

So here we can now create a new class
in this expense item dart file,
and that's the expense item class
which extends state less widget again,
state less widget because there won't be any internal data
that would need to be managed.

And of course here we again need the build method
and we also need a constructor expense item
because we again need to get some input data here.

We need the expense data
that should be displayed on the screen.

For this, I'll add a new property to the expense item class,
which should be of type expense.

And of course for that,
you again must add the import to this expense dart file
in the models folder in your project folder
so that I can add my expense property,
which is of this type here.

And then in this constructor,
we can set this to mix things up here
with a positional argument though you could also wrap this
into curly braces to turn it into a named argument.

Positional arguments by default are always required,
hence the required annotation is not needed
and not supported here.

And we still of course also need to add a second
acceptor parameter, which is a named parameter
and hence wrapped between curly braces.

And that would be this key parameter,
which is then forwarded to the parent class
to the stateless widget class.

With that, we can now output our widget,

our combination of widgets down here.

And here for this expense item,
you can of course definitely play around
with the different widgets supported by Flutter
with the things you'll already learn about
throughout the course.

This is a great exercise, so definitely do that.

Here for this widget,

I chose to start by wrapping everything
with the card widget,
which is a widget we haven't seen before.

This is a widget which I simply use for styling purposes
because this will give us a nice card look.

It will basically put the content that we pass to card
into a container that's a bit elevated
that has a slight shadow behind itself
to make it stand out a bit more.

I can show you how it looks by setting the child parameter,
the named child parameter on this card widget
and set this to the text widget.

And now here I want to access expense title.

This is not the final definition of my expense item,
but this allows me to show you
how this card widget will look like.

So here I output expense title,
and now we can use expense item
in the expenses list dart file by going down here
and removing this text widget in the item builder function.

And instead we wanna create our expense item here.

If I use this auto suggestion menu here,
it again automatically adds the import for me.

Otherwise you must add this import manually to your project,

the widgets folder expenses list,
and then expense item.dart.
And with that added down here,
we can create an expense item by instantiating it,
and I then wanna pass a single expense to it.
So I access my list of expenses that I get here,
then use the index again with the square bracket notation
to access a single expense.
And that's then the value
that should be passed as a value to expense item.
Since there, we expect to get an entire expense.
And with that, if I force a reload,
you now see the titles again.
But if you look closely,
you see that there is this slight drop shadow
and that's this card look.
It doesn't look too amazing here
because we got no other styling and no other widgets,
but I want to have this elevated look here
to make these items stand out at least a little bit.
A card also automatically adds
some top and bottom margin
so that we have some spacing between those items.
It's a bit hard to see here,
but it's a subtle effect
that will make the items look better,
especially once we're done adding all the other widgets
that should be added here to expense item,
which is what I will do next.
So here at expense item, we now have this card.
As a next step, I want to add some padding to this card.
Now, card does not have a padding parameter

that could be set.

It has a margin parameter with which we could change or extend the default margin that is added anyways, but it doesn't have a padding parameter.

And if you recall, padding simply is this internal space, so some spacing which I want to add between the text and the border of this card.

Now you can still add padding by refactoring this text widget and wrapping it with the padding widget.

This then allows us to set some padding.

And here, I don't want to add the same padding in all directions,

instead, I'll use the special symmetric constructor function on edge insets, which is this utility class provided by Flutter

that's used for creating margins and paddings.

And here, I wanna set my horizontal padding to let's say 20, and then my vertical padding to let's say 16.

And with that, you now see that there is some spacing between the borders of this card widget and the actual content, the text, which already ensures that this looks a bit nicer, I'd argue.

Now of course, that's still not all though.

I don't just wanna have the text in there.

Instead, the goal is

to later have the text here in the first row, so to say, and to then have a second row of data inside of that card which displays us, for example, the amount, but also the date and category.

So therefore here, since I wanna have two rows,

we need the column widget again.

And here it's column and not list view

because it doesn't need to be scrollable

and list view makes it scrollable by default,

but we don't need this here, it's not a long list.

And we also don't need that performance optimization

because it is a list that's guaranteed

to consist of only two rows inside of this item.

Of course, we might have multiple items,

but we are optimizing those

by displaying them with list view.

Inside of a single item,

since we will only have these two rows,

we don't need that list view.

So here we can then set children

and children here should be set to multiple widgets.

And the first widget will be the text widget again,

which indeed will output the expense title again.

We'll change its styling later,

but in general we want to output the title

as the first row in this item.

But now, we're going to add more widgets.

I will actually add another row,

technically a third row if you wanna call it like this,

by using SizedBox to add some fixed spacing,

a fixed height between the text

and the actual second row,

the actual visible second row, you could say.

Instead, we could also use margin

wrapped around the first or second row,

but this is simply a bit quicker.

Now I want to add that actual second row,

and that will now actually not just be a single widget
as we have it here in the first row
where it's just the text,
but instead, it will be the amount,
and then to the right of that,
it will be a combination of the category
or an icon representing that category and the date.
So therefore here we need the row widget
because that's what we can use if we wanna display
multiple widgets next to each other horizontally.
For that, we add children here.
And the first child here is now another text widget,
which is in general,
a widget you'll use a lot when building Flutter apps.
Because here I want to output the amount
that belongs to this expense.
For this, we can of course access expense amount like this.
However, you see I get an error here
because this is actually of type double
and the text widget wants a string.
Now we can convert it to a string by calling `toString`,
but I will actually call a different method.
I will call the `toStringAsFixed` method,
which then wants an argument, which I'll set to two here.
But what does this do?
Well, `toStringAsFixed` simply ensures
that a value like 12.3433
would simply be displayed like this
so that it is limited to two digits after the comma.
That's what this does.
So it's simply a stylistic choice here to make sure that
if users would enter amounts

with too many decimal places, we only display two.

I actually also wanna display the dollar sign
in front of this amount.

And for that, we can wrap this with single quotes,
but now this would all be treated as text
and therefore I'll instead use this injection syntax,
which I've shown you before in the course
with the dollar symbol,
which now won't be displayed as a dollar symbol,
but which instead tells Dart
that we want to inject a value into the string,
but here the value is actually the result
of a more complex expression instead of a single variable.
And therefore here you actually have to extend this syntax
and add a opening curly brace,
and at the end, a closing curly brace.

And this simply tells Dart
that it should take this value here as one value
that should be injected into this string.
Now, as I mentioned, I wanna output a dollar sign
in front of this string
and this dollar sign won't be output,
instead, that belongs to Dart.

If I do wanna output a dollar sign,
I instead have to escape that character as it's called
by adding a backwards slash
in front of this special dollar sign here.
This backward slash now won't be output,
instead, this now tells Dart that this dollar sign
after the backward slash should be output as text
and should not be treated as this special injection syntax.
So here we won't see two dollar signs on the screen,

but instead this second dollar sign
belongs to this injection syntax
and this first dollar sign will be output
because we're escaping it as it's called.
And with that,
we should display this amount below the title.
Indeed, if we save this,
we should already be able to see this.
Of course, we'll soon make sure
that the title won't be centered anymore,
but in general, this doesn't look too bad.
We have the title, we have the amount,
and we have this dollar sign in front of the amount.
Now we're still not done
with the expense item widget here though.
We're now outputting the title and the amount,
but there's more information that should be output.
In this row here,
so in the same row as we're outputting the amount,
I also wanna output information
about the category and the date.
And for this, I'll add a row inside of this row
because I wanna group category and date together.
They should be output next to each other,
but they should be pushed
all the way to the right in this card here.
So the amount should stay on the left
and the other information,
the other two items should be displayed
next to each other horizontally, but pushed to the right.
Hence we have to put them into a separate collection,
you could say a separate group of widgets.

And since I want to output category
and date next to each other horizontally,
I will group them together with row.
But I'm doing this so that between this amount text
and this row, I can add another widget
and that will be the special spacer widget,
which is also built into Flutter
and which can be created with const.
Spacer is simply a widget that can be used
in any column or row to tell Flutter
that it should create a widget
that takes up all the space it can get
between the other widget between which it is placed.
So text gets all the space it needs to output the text.
And this row will get all the space it needs
to output its child items.
But spacer then will always take all the remaining space,
hence pushing text to the left and this to the right.
And now with that, we of course can and should
add children to that inner row.
And here we now want to have this category
or this icon that represents the category
and then next to it, the date.
Now for the icon, we can use the built-in icon widget,
which can be used for displaying an icon.
And for the moment, I'll just use a dummy icon here.
We'll soon use a dynamic icon,
which will depend on the category.
For the moment, I'll just use any icon here.
Like for example, let's say the alarm icon.
And then next to that, I'll add a sized box
to add some distance between this icon and the date

with a width of eight, for example.

And next to that,

I have my text that should output the date.

And we can also add const here

though that will soon change

since soon this won't be a hard coded dummy icon anymore,

but for the moment, it can be const.

But now for this text here,

I want to output the date of my expense.

And for that, we can of course access expense date.

But this would be of type date time, not of type string.

And it also wouldn't be formatted at all.

Now we'll soon take care of formatting.

For the moment, we'll just convert it to a string

by calling to string.

This will make sure

that the date will be output as a text date.

Well, and with all of that, if you save this,

you should see all this information on the screen.

You see the title here, you see the amount,

and of course you also see this icon for the category.

And then next to it, the date which looks ugly as mentioned

because it's not formatted at all.

And indeed, that's the next thing we'll take care of.

We'll make sure that the icon is actually chosen

dynamically, that it depends on the category of the expense,

and we'll make sure that the date is formatted

in a more human readable way.

9. Using Icons & Formatting Dates

Now for both the dynamic icon

and the formatted date, I'll actually go back

to my expense start in the models folder,
because in there I want to tweak the expense class
to make outputting a formatted date a bit easier.
And I want to add a new value to this file.
So not to the expense class, but to the file as a whole.
Here I want to add a constant value called category icons,
or the constant is called category icons.
And the values should be a map where the keys will be
category enum values.
And the values
of those keys in this map will be the actual icons
that should be displayed for the different categories
so that in this expense dart file,
we don't just store the definition
of our expense value objects,
but instead we also store the available categories
and the icons that belong to these categories.
You could split this across multiple files,
but here I'll keep it all in one file.
So here my keys are these different enum values like
category food with the dot notation here.
And then the value set with a colon here in a map
as you learn before in this course would be an icon.
And here I'll use Flutter's built-in icons class,
which of course must be imported.
So material dot dart from Flutter must be imported.
And then here for food, I'll use the lunch dining icon.
And of course you can browse through all those icons
and pick the icons you like most.
But I'll use this one for the food category.
Then for the travel category, I'll use icons,
flight underscore takeoff.

For the leisure category, I'll use icons dot movie.

And for the last category, the work category,

I'll use icons dot work.

So these are my icons

and I'll soon use them here in expense item dart.

But before we do that, I want to tweak this expense class.

This class at the moment is simply defined

to define the data structure of our expense objects.

And this is of course very useful

because that allows us to create such expense objects

with ease and it gives us auto completion

and type safety when working with them.

But of course, you can not just store properties in classes.

Instead, you can also add methods to classes.

And here we could add a method called Get Formatted date,

though if you find yourself defining methods that start

with get as a name

or that are in the end all about transforming one

of these property values, you could, instead

of creating such a method, also create a so-called getter

for that you use the special get keyword built into dart,

and then the name of your getter,

which now should be named like any other property,

for example, formatted date.

And you should also define your return value.

So the type of value that will be produced by this getter.

And you skip these parentheses.

Instead, you just have opening and closing curly braces.

And between those curly braces, you return your value,

which should depend on your other property values.

And here I wanna reach out to my date

property and format that.

Now formatting dates manually in Dart is quite cumbersome.

And for that reason we'll use another third party package

here, the Intel INTL package

for which you can simply search to find it on pub dev.

And as always, you can learn more about this package [here](#).

It's a quite powerful package that can help

with the localization as well.

But here we'll use it to output a formatted date.

And for that we'll just add it to this project,

a running Flutter pub at Intel.

And with it added in this

expense dart file, we can import it just

as we import it Flutter

or the UUID package by importing intel.

And then here the Intel Dart file.

That's what we need here.

And now with that package imported,

we can use the date format class, which is

provided by this Intel package.

And there the YMD constructor function

and will store the value

of calling this constructor function in a final variable

that could be called formatter.

And YMD simply creates a formatter object,

which we can then use to format dates.

So it's a utility object you could say, created with help

of the date format class,

and this special constructor function

provided by the Intel package.

And this YMD constructive function defines

how the date will be formatted.

And indeed date format supports many constructive functions

that could be used to format dates in all kind of different versions.

Y stands for a year, M for month and D for day, and YMD used like this is just one of those predefined formatting options.

So now we can use this formatter object down here in this getter.

And on that object you can call the format method and then pass your date to it.

And this will return a string containing the formatted version of that date.

So the more human readable version.

And now with this formatted date added

and this icon map added here,

we can use this information here in expense item.

And for the icon here in this inner row, instead of always using the hardcoded alarm, we can now reach out to our category icons map, which we have in that other file,

and use this square bracket notation here

to access a specific item in that map.

And since in that map we use Enum values as keys.

We also have to use such an enum value as a key

to access an icon, which is no problem

because in our expense object with which we're working here,

we have this category property,

which stores enum values of that category, enum.

Now we also have to remove const here though,

because this isn't a const value anymore.

Instead, this is generated dynamically well

and forward a date.

We now access expense, dot formatted date.

You don't add parentheses here
because it's a getter, not a method.
If you did define formatted date as a method instead,
which you could also do, you would call it as a method.
But here we're using this getter version.
With all that done, this now looks way better
because now we got different icons for different expenses.
And as you can also see, we got formatted dates here,
which are way more human readable.

10. Setting an AppBar with a Title & Actions

So we're making progress,
but of course, for example,
the styling still isn't where it should be.
But we'll take care about this later
because I think the more important thing right now
is to make sure that we're not limited
to those dummy expenses, which we created here.
It's good to have them to create that list
and these list items.
But of course, ultimately the goal is
that users can add their own expenses.
That is what this app will be all about.
And if you recall the final version of the app,
you might also remember
that there we had this toolbar at the top, this AppBar,
this plus button at the top here, which we could tap
to bring up that modal overlay that allows us
to enter new expenses, to insert new expense data.
So of course we have to work on that modal overlay,
but first of all, we have to add this toolbar at the top.
Now for that and the expenses Dart file in this column

where we want to have the chart,
and then also this list of expenses,
we could of course add this Toolbar
with the Add button here
and try to use a Row widget, for example.
We could do that, but having such a toolbar at the top
is such a common requirement
and is often needed for so many different things.
For example, also to output a title
that Flutter actually has a more official support for that
and does not force us
to build our own here instead of this column.
Instead, this scaffold widget is not just there
to give us this white background and some basic styling,
but besides doing that,
it also supports multiple other arguments
besides the body parameter, which we set
to the main page body.
It, for example, also has this AppBar parameter.
It also has other parameters which we'll
see later in the course.
For example, drawer for adding a side drawer.
But as mentioned, we'll see
that later in the course in another course section.
But here it's the AppBar parameter, which we want to set
because AppBar can be used
to add such a bar at the top of the screen.
AppBar wants a PreferredSizeWidget, which simply means
that it only takes certain kinds of widgets.
And typically you use the AppBar widget here,
which is another widget built into Flutter.
Now the AppBar widget then takes a couple

of configuration parameters as you see,
and as always, feel free to browse through them.
But most importantly for us here,
it offers an actions parameter, which we can set
and actions wants a list of widgets,
and this is typically used
to display buttons in this top AppBar in this toolbar
at the top of the screen.

Now here you could add a text button or an elevated button
or any other kind of button you already saw in this course,
but here I'll use the icon button instead,
which is another built-in type of button,
another built-in widget you can use in Flutter
to display a button
that only contains an icon.

So not text and icon, but just an icon.
For that, icon button needs the icon named parameter
to be set and it should be set to a widget.
But of course here typically you use the
built-in Icon widget.

And here I'll use the built-in add Icon widget like this.
Now this icon widget is actually const,
so we can add const here, but that's not all we have to set.
Instead, since it's a button,
we should also set onPressed here
and onPressed just as on the text button,
which we already saw before in the course, wants a function
as a value, an anonymous function, or a function defined
before in your class.

Here, I'll for the moment,
set it to an empty anonymous function.
Now with that, if you save this,

you have this AppBar here at the top,
which might look like this or which might look like this.
It simply depends on whether you're using the Material
2 or 3 look, and at the point
of time I'm recording this, Material2 was still
the default, though no worries.

I built the entire course
with Material3 in mind.

But if you still had that other look with that blue bar,
you can go to your main Dart file
and simply add this line
to your MaterialApp constructor function
and set the theme parameter to ThemeData.

And there set useMaterial3 to true,
and I'll get back to theming later in this section,
but you can do this now already
to get this more modern AppBar look here.

Either way, you have this plus button here,
this icon button, which we just added, of course.

What you can also see is that now
that the AppBar was added, Flutter made sure
that all the content,
including the AppBar, fits better into this screen.

It basically automatically reserves the space needed
for the camera and so on to not be occupied by widgets.

And instead make sure that your widgets like the AppBar
or this chart widget
or this list widget all go below this space up here,
which it reserves for these device inputs and so on.

So we get all of that just by adding AppBar.

And as mentioned, actions are important,
but not the only thing you can set on AppBar.

Instead, there are many other parameters as well that can be set. For example, you could change the background color, but again, I will come back to styling and theming later in this section.

But you can, for example, also set a title here, which again is a widget, typically a text widget.

And here we could set this to Flutter ExpenseTracker and turn this into a constant widget like this.

So with that added here, we also have this title here, and that's how you can add such an AppBar.

Again, we'll take care of styling later.

As a next step,

I instead wanna make sure that pressing this plus button actually does something and that it opens this modal overlay, which then allows us to add new expenses.

11. Adding a Modal Sheet & Understanding Context

So let's now make sure that clicking this plus button in this app bar here does open this modal overlay at the bottom, which then does allow us to enter new expenses.

For that, of course, we have to work on this function here, which we assigned to the onPressed argument of this icon button

because that is the function that will be triggered when the button is pressed.

And we could edit it here inside of this build method.

So we could work right inside of this anonymous function here.

Or alternatively, what I will do here,

we can add a new method
to this `_ExpensesState` class.
And this method here could be called `openAddExpenseOverlay`
or anything like that.
The name is, of course, up to you.
It's a method that doesn't need to return anything,
hence its return type is `void`.
And inside of the curly braces,
so between the curly braces,
we're going to add the logic for showing this overlay.
But before we do that,
we want to assign this function as a value to `onPressed`
by pointing at it like this
as you learned without parentheses
since we don't want to execute the function here,
instead we want to use it as a value for `onPressed`
so that Flutter internally, so to say,
can execute this function here
whenever this button is pressed.
Now inside of `openAddExpenseOverlay`,
I now want to execute some code
that does open such an overlay.
And thankfully here we can use a built-in feature.
If we type `show` here, we can get a long list
of built-in functions
that are all provided by Flutter
that we can call to show built-in UI elements,
like a date picker or a dialog box.
And we'll see some of these things,
for example, the date picker a little bit later.
Now here it's actually the modal bottom sheet
which we want to show.

So it's show modal bottom sheet, which we want to execute inside of this `openAddExpenseOverlay` function.

And this show modal bottom sheet function here is provided by a Flutter.

And hence it's available

because of this material dart import,

which as you can tell by now,

really unlocks many things, many widgets

and also, for example, this utility function.

Now this function will dynamically add a new UI element, such a modal overlay when it's being executed.

And since I created it here

by using the suggestion whilst typing it,

my code editor automatically set it up

to use two named arguments, context and builder

to which we have to pass values

when calling show modal bottom sheet.

Because these are two required arguments

that must be set when using this function.

Now, context here wants a build context value.

For example, this context here,

which we get automatically as an input to the build method.

So which is passed to the build method by Flutter.

Here we're outside of the build method,

and yet there actually is a globally available

context object, a context property

that's made available by Flutter automatically

since we're in a class that extends state.

When you're in a class that extends state,

Flutter automatically adds a context property

to your class behind the scenes, so to say.

It's provided by this state parent class.

And this context property that's made available by Flutter can be used as a value for this context argument that must be set on show modal bottom sheet.

But for that, it's of course, important to understand what this context thing is all about.

And in the end, you can think of this context object, this context value as some kind of metadata collection.

An object full of metadata managed by Flutter that belongs to a specific widget.

So every widget has its own context object and that contains metadata information related to the widget and very important, related to the widgets position in the overall UI, in the overall widget tree.

So this context value which we get here holds information about this expenses widget in the end and its position in the widget tree as you'll also see if you hover over this context value.

Now this object, which is full of metadata, is required by show modal bottom sheet, which is why you have to pass it as a value for the context argument here.

And the second argument that must be set is another builder argument.

And whenever you see an argument called builder, it basically always means that you must provide a function as a value.

And here it's a function that should return a widget and that automatically gets one input value passed into it by Flutter, which we'll call this builder function basically

when it then tries to show this modal bottom sheet.

So here we can set up a function,

which gets a context value.

And here I name this context parameters ctx

so that I don't clash with this context.

I could use the same name,

but technically it's a different context

because it's the context of this modal bottom sheet

that will be created.

So the information object related to that,

instead of the information object

related to the expenses widget,

and hence I choose a different name here.

And inside of this function body,

you now to return your widget

that should basically be displayed

when Flutter renders this modal bottom sheet

that it's opening up.

And therefore, here we can again use

this very short notation using this arrow syntax

since here we return that widget that should be shown.

And to get something on the screen quickly,

I'll again use the text widget here as a placeholder

and just say modal bottom sheet

and add a semicolon at the end here

and a comma here for formatting reasons.

And now, this is the code that we must execute in here

to show some modal bottom sheet.

And, of course, the text widget will soon be replaced

with the actual content that should be in that modal sheet.

So with that, if you save everything

and you click this plus button,

you should now get such a modal sheet at the bottom.

At the moment, just a tiny one

because there isn't a lot of content in there,

but there is this overlay,

which closes if you click on that backdrop

that also opened up.

You see that the background gets grayed out

when this sheet at the bottom opens.

And if we click on that backdrop, the sheet closes again.

So that's how we can open and close it.

And now, of course, we have to work on the content

inside of that modal bottom sheet,

and we also might want to add

some additional configuration to it

to make sure that it behaves correctly.

12. Handling User (Text) Input with the TextField Widget

So let's now replace this text widget

with the actual widget

that should be shown in this ModalBottomSheet.

And for this, I'll add a new file

because this widget,

which I wanna show in this bottom sheet, will be this form,

this combination of input elements

that can be used by the user to describe a new expense.

And since I'll have a couple of widgets there

and some logic attached to them,

I wanna move that to a new file.

So here I'll create a new file in that widgets folder,

which I'll name, new_expense.dart.

So that's my file name.

And in there we wanna create a new class,

the `NewExpense` class,
which of course extends a widget class.
And here it'll be the `StatefulWidget` since I will need
to manage some internal state here in the future.
So therefore here we're extending `StatefulWidget`.
We need to add the `createState` method here,
and of course, the constructor function
for this new expense widget.
And at the moment here, I'll just accept
and forward the key argument.
And then we add this second class
that's always added to your `StatefulWidgets`,
the `_NewExpenseState` class, which extends `State`,
which is linked to `NewExpense` like this.
And here for the `createState` method,
we can also replace `StatefulWidget`
with `NewExpense` like this.
And here in the `NewExpenseState` class,
we of course must add this `build` method.
And here in `createState`, we have to return a new instance
of `NewExpenseState` like this.
And here in the `build` method, we now of course have
to return our combination of widgets
that should define this form, which can be used by the user
to enter the expense data.
For that, I'll return a widget here,
and I'll start by returning a `Padding` widget, actually,
since I want to have some spacing around this form
so that it doesn't sit right on the edges
of this `ModalBottomSheet`.
So here we add padding
and set padding to `EdgeInsets`.

And then here for the moment, I'll set this to all(16).

And I'll add the const keyword later
once we finalize this widget combination
and we know where exactly we can add it.

So here I added my padding.

Now we also must add the child argument
and then of course, define which widget should be wrapped
with the padding.

And in the end, in this finished form
that should be shown on the screen,
I'll have multiple rows of input elements,
you could say.

I have this title input element in the first row,
and then in the second row I have the amount
and the date input element.

And then in the third row at the bottom,
I have this dropdown, which can be used
for choosing a category.

And then next to it, my buttons that can be used
for canceling input or for submitting the input.

And therefore, since we want to have multiple rows,
it again screams for the column widget.

And we can use column instead of list view here
because we don't need scrollability right now.

And we also don't need to generate those items dynamically.
Instead, we know exactly how many rows we'll have
and therefore we're good to go.

So with that, we can set our child widgets in this column.

And I will start with the input field for the title
of the expense that is about to be added.

And for that we can use the built-in TextField widget
provided by Flutter,

which will display such a text input element.

Flutter has a bunch of built-in input elements.

For example, later we'll also use the dropdown button for displaying this dropdown.

But here it's TextField for allowing the user to enter some text.

Now the TextField widget has many parameters that can be set.

Many niche parameters, which you will rarely need, but also some very useful parameters like for example, the max length parameter, which you can use to set a maximum amount of characters that can be entered into this input field.

For example, here for our expenses, we could say that the title of a single expense shouldn't be longer than 50 characters, and hence we can enter a value of 50 here.

In addition, we could also set the keyboard type here to then use the built-in text input type enum, which is provided by Flutter to control which virtual keyboard should be opened when the user taps into this text field.

So that we, for example, open a keyboard that's optimized for entering email addresses or standard text, which is the default setting anyways though, and which we therefore don't have to set explicitly.

And nonetheless, it is important to know about this feature.

Now here for this text field, I don't need that, but I do want to add a label to the text field to tell the user which kind of value should be entered there so that the user doesn't have to guess.

And the label can be added and that's a bit strange

by setting the decoration parameter on the text field.

So I guess you can think of this label text as kind of a decoration around the actual text input field.

So decoration then wants a input decoration object created with the built-in InputDecoration class provided by Flutter.

And InputDecoration offers a label property.

So that's how you add a label to a text field, you the decoration parameter to an InputDecoration object.

And on that object you can set the label property and the label wants a widget as you can see.

So therefore here we can use the Text widget, which is the widget you will typically use for such a label to give this text field a label of title, for example.

With all of that, we can add const here to get rid of the blue squiggly lines and also add it here in front of Edgelnsets.all.

And we should have such a text field.

Now, this is not the final form.

We'll add more to it later.

And at the moment, we're also not handling any input into this text field, but it's a first step and I wanna make sure that it's this form, this basic form here that will be displayed in this bottom sheet when it's opened.

And for that, of course, we have to go back to expenses to this code where we open the ModalBottomSheet, and instead of showing this tummy text widget here, I want to create my NewExpense widget instead.

And we can add const in front of this here.

With that, if we save this and I force a reload,

if I open this again,
you see this looks totally different now.
Now it looks better.
It occupies more space
because of that color, which we added in there,
which automatically in the end tells Flutter
to reserve more space for it.
We got the padding around the text field.
And of course, most importantly, we got the text field.
We got the text field with that label that we set,
and with a hint related to the max characters we set.
And of course, we can tap into it
and this soft keyboard should open up here,
and we can use that to type or here when using the emulator,
we thankfully can also use our physical keyboard
to enter our title.
So that's how we could enter a value here.
Of course, at the moment, we're not using
that value in any way, and we also can't submit the form.
So therefore, before we add any other input fields,
let's make sure that we do instead save that entered value
and that we can submit that form
and add a new expense with that title.

13. Getting User Input on Every Keystroke

So, back here in `new_expense`,
let's make sure that we store the data entered by the user
and we then also use that data when the form is submitted,
so when this input field is submitted in the end.
And for that, of course,
we'll also need a button that allows us to submit this.
Now, first things first.

I wanna start with storing the value entered by the user.

And when it comes to storing that value,
we got two main approaches for doing that.

The first approach is to add another parameter
to the text field,
and that would be the `onChanged` parameter,
which allows us to register a function
that will be triggered
whenever the value in that text field changes.

For example, because the user pressed a key.

And `onChanged` wants a function
that will receive a `String` value
and that string value that's received
will be the value entered into the text field.

So, therefore, we could add a method here
to `NewExpenseState`, a method that returns nothing,
and could be called `_saveTitleInput`.

And here, we get our `inputValue`,
which is of type `string`, as mentioned before.

And now we can pass `saveTitleInput` as a value to `onChanged`
like this, not executing it, just pointing at it.

With that, we have to remove `const` though,
since we're setting this function
as a value for `unchanged` now.

We could add it here in front of input decoration.

And inside of `saveTitleInput`,
we can now use that `inputValue`,
which is passed in automatically by Flutter
whenever it calls this function,
which, as mentioned,
it does whenever the value in the text field changes.

And we could now add a variable to our state here,

enteredTitle, or `_enteredTitle`,
which initially is an empty string,
because initially nothing has been entered.
But now here, whenever this is triggered,
we can set `_enteredTitle`
equal to the entered `inputValue`, like this.
We don't need to wrap this with `setState` here,
because I'm actually not using entered value
anywhere in my UI code.
So, the UI doesn't need to update
just because I am storing
the value entered by the user in some variable.
Instead, I'm storing that value in some variable
so that we can use that variable
with that stored latest value entered by the user
when the user submits the input.
And for that, we need to add a button
that allows the user to submit the input.
So, after the text field,
still inside of the column, of course,
I wanna add my buttons.
And for that, I'll add a row here
since I'll have more than one button eventually,
which takes multiple child widgets, of course.
And for the moment here,
I'm only interested in adding one button here,
an `ElevatedButton`, let's say,
on which we have to set the `onPressed` argument
pointing at a function
that should be triggered when the button is pressed,
and then the child widget
that contains the content of the button.

So, for example here, a text that says Save Expense,
and this text here can of course, be a const value.
And then onPressed, we now wanna use this entered title
to add a new expense.
Now, we'll do this eventually.
At the moment, I'll simply print this entered title
to see whether A, this code is executed,
and B, we get access to that latest value that was entered.
It should work because we're saving the latest value here
on every keystroke in enteredTitle,
so therefore, we should have the latest value
entered by the user stored in that variable.
And therefore, if we then,
at a later point of time, reference this variable here,
when this button is pressed,
we should be able to print that latest entered value.
So, with that, if you save that
and maybe force a reload,
you should be able to enter some text here like test,
and click Save Expense.
And if you now go back
and you open this panel here
in Visual Studio Code
and you dare go to the DEBUG CONSOLE,
you should see this Test value somewhere there,
or whichever text you entered into your input field.
So, this works.
We do have access to the enteredTitle,
and that's one way of getting hold of that value.

14. Letting Flutter do the Work with TextEditingController

So reacting to every key press

and updating and storing the current value of the text field
in some variable
is one way of handling and storing the user input.
But of course, it can be a bit cumbersome
to manually store all the input
for all the text fields
that might make up your user interface.
And that's why Flutter gives you a convenient alternative.
Nonetheless, it's also important
to know about this approach
because it's also a valid approach
that you might see in many projects you're working on.
But the alternative you get
does not require you to add this function
and this extra storage variable.
Instead, we'll add another property,
which can be a final property now
to this `NewExpenseState` class.
And this is a property
that could be called `titleController`.
Now, the name might be a bit strange,
and of course it is up to you,
but I chose `titleController` here
because the value which will be stored in this property
will be an object that is created
with help of the built-in `TextEditingController` class,
which is a class provided by Flutter.
Now, this creates an object
that's optimized for handling user input, you could say,
an object that can be passed as a value to text field
to let flutter do all the heavy lifting
of storing the entered value and so on.

Now, when using `TextEditingController`,
there's one thing that's really important
to keep in mind though.

When you create a `TextEditingController` here,
you also have to tell Flutter to delete that controller
when the widget is not needed anymore,
so for example, when the modal overlay is closed,
because otherwise this `TextEditingController`
which you create here
would live on in memory
even though the widget is not visible anymore.
That's why when using `TextEditingController`,
you should also always add the special dispose method
to your state class here.

So a method called `dispose`,
which returns nothing,
and calls `super.dispose`.

And before calling `super.dispose`,
you should reach out to your controller,
and on that controller,
call the `dispose` method,
so that you tell Flutter
that this controller is not needed anymore.

Otherwise it will not be deleted,
and ultimately your device memory
might be taken up by all these unnecessary controllers,
and that could cause your app to crash.

So therefore, you should always dispose
of these `TextEditingControllers`
you might be creating.

But what's the idea behind this controller now?

Well, with it created,

you can go to your text field
and get rid of this onChanged parameter,
which you now don't need anymore.
And instead you can set a controller parameter.
And you set that to the controller
that should be responsible for this text field.
So if you have multiple fields,
you need multiple controllers.
And you can now also use this titleController here
in the function that's triggered
when this button is pressed.
Here, you can access your titleController,
and then on that object,
you will have a text property,
which, as you might guess,
simply contains the text
the user entered into the input field.
And therefore you don't have to manually take care
about storing that input.
Instead, Flutter does it for you.
And with that, of course,
if we go back and reopen this
and enter Test or Test 2,
and click Save Expense,
and we go back to that debug console,
we now see that output here in the console.
And that's therefore the approach
we'll use in this section now.
And therefore, since it's a bit less work to do
and we will add more text fields
and I don't want to manually manage all the inputs,
I will use this controller approach in this section here.

15. Time to Practice: Adding a New Input

So now that we have a basic understanding of how we can add text fields, here's a little exercise for you, a small challenge for you.

I want you to add another text field to this column in a separate row.

And that text field should allow the user to enter an amount, so the amount that belongs to the expense that is about to be created.

For that, of course, you wanna make sure that you use a soft keyboard type that's optimized for entering numbers.

You wanna add an appropriate label to this new text field.

And of course, you also should make sure that the value entered by the user is stored and is also printed here down here in this button.

And speaking of buttons, I also want you to add another button to this row of buttons here, a text button which allows the user to cancel editing and which will use in the future to close this modal overlay.

Now, you don't need to add the logic for closing just yet because you haven't learned how to do that, but definitely add the button and add that text field and therefore practice what we learned.

16. Exercise Solution

So did you succeed?

Let's now try it together.

For that, I'll, first of all, copy that TextField, which we had here for the title, and adjust it for my amount here

by replacing the text with the text amount.

The `maxLength` is actually not needed anymore,

but I'll set the `keyboardType` parameter,

which I showed you before,

to input type, or `TextInputType.number` like this.

And of course, I also want to create

a new controller here,

an `_amountController`,

which is also a `TextEditingController`,

and which also must be removed

when we remove this widget.

So here in this dispose method,

which is called automatically by Flutter

when the widget is destroyed,

I will use my `_amountController`

and call `dispose` on that.

Like this.

And with that, Flutter will manage that value for us,

at least as soon as we assign the `_amountController` here

as a value for controller on the `TextField`.

And we can then use it here on this button down there

to also output the `_amountController.text`.

Now please note that this will indeed be text.

So a string value, even though a number is entered,

if we want to treat it as a number

and use it as a number,

we'll have to convert it manually, thereafter.

But that's something we'll do later.

For the moment, I'll just output it here.

And I also wanna make sure that this second button,

which I mentioned, is shown here.

And this will be a `TextButton`,

where on press I currently won't do anything.
I just add an empty anonymous function here.
And as a child, I'll set the text
which should be displayed.
And here that's simply 'Cancel,'
which can be a constant piece of text.
Now with that, we added the two things
that should be added for this task.
There's now just one additional thing
I want to do to this TextField,
to this amount TextField,
which was not part of the task.
So you're done, you're good.
But there's one thing I wanna do.
In this InputDecoration object here,
I wanna set the prefixText property here
to a dollar sign.
And since this is this special character
with this special meaning,
I'll escape it by adding this backward slash
in front of it.
And I'm doing this here, followed by a wide space
so that we always have this dollar sign
in front of the amount field
because, well in the end, it is a dollar amount
that should be entered here.
So that's some additional configuration
to make sure that we tell the user
which kind of value they are about to enter.
With that, if we save this and we force a reload,
if we open this modal overlay,
we got these two buttons here at the bottom.

That's great.

The Cancel button, of course, doesn't do anything, but we'll change this soon.

But if I enter a value here in the title and here in the amount, that both works. And you also see how the keyboard changes because of this keyboardType that we set here on the second input field.

We get this number optimized keyboard here, and we can, therefore, enter a number.

If we now click Save Expense here, this should both be stored.

And indeed, if we take a look at the debug console, we can see both values here.

So that's how we can add a number input field here.

17. Closing The Modal Manually

Now, being able to enter all these values is great, but let's make sure that now this cancel button also closes this modal overlay when tapped.

And to achieve this, we have to go to this function here, which is assigned to this cancel button.

And of course, we could also convert this into a method that belongs to this class, and then just point at it.

But here, I'll work with this anonymous function, just like that.

But inside of this function, we now want to use another built in class provided by Flutter, the Navigator class, which we'll see, again, a little bit later in the course, once we dive deeper into navigation

and how we can switch between different screens.

And this navigator class has a pop method,
which we can execute,
which wants the current context as an argument.

So this context value,
which we get automatically here in this build method.

So this is passed in by Flutter,
and we have to kind of forward it to the pop method.

And pop simply removes this overlay from the screen.

So if we save that
and I click cancel, you see the overlay closes.

So that's pretty good.

The next step now is to make sure
that this text field widget here
doesn't take up its entire row,
but that it instead sits next to another
input-like element, not a text field actually,
but some other widget, which we will kind
of treat like an input element,
that then allows us
to select a date that should be attached to this expense.

18. Showing a Date Picker

So it's time to add a date input, so to say.

And for that, I'll add a row widget here
below this first TextField
because this amount TextField
actually doesn't need an entire row.

The amount is typically pretty short.

We don't need to use the entire row here.

Instead, I wanna use a part of that row
and the other part should be used by the date input.

So therefore here, this row takes its children parameter
and the first child that's added here is the text field.

The second child after the text field
will be that date input.

But here, we don't have a built-in input that we could use.
Instead, we kind of have to build our own input here.

But thankfully, as you'll see,
that will be quite straightforward.

No worries.

To add this date input here,
I'll start by adding a SizedBox here
to add some spacing
between this amount TextField and that date input,
let's say 16 pixels.

And then next to that,
another row which I wanna use
to group two other widgets,
which then together form the date input together.

And since we here have a Row inside of a Row
and that would cause problems,
we should already wrap this with Expanded,
and we will actually have to do the same for the TextField
because TextField turns out to be a widget
which has problems

with being placed in a Row like this
because it also wants to take
as much space horizontally as possible,
and the Row by default
doesn't restrict the amount of space that can be taken,
and that therefore causes problems
when Flutter tries to render this combination on the screen.

Hence, we also should wrap TextField with Expanded

to make sure that it takes as much space
as it can get on the screen, but nothing more.
So we need to wrap this around TextField
and also around Row here.
With that, inside of this inner Row,
I want to have a Text widget,
which we'll use to show the currently selected date,
so for the moment I'll just say Selected Date here
as a placeholder.
And after this text field here,
I want to display an IconButton,
which can be pressed to open a date picker,
so an overlay that can be used for selecting a date.
This IconButton here needs an icon.
And for this, we can create a new const Icon here
with Icons.calendar_month,
which is a fitting icon I want to use here.
And in addition to setting the Icon,
of course, I also wanna set onPressed
to make sure that I register a function
that will be triggered when this Icon button is pressed.
And I'll get back to that function logic in just a second.
With all that though, if you save that and force a reload
and you click this Add button,
you should see something like that.
It's not 100% finished yet,
but you see that the amount
and this date input combination here sit in the same row
and take up an equal amount of space here.
Now, indeed here for the date inputs,
it would be nice if they would be pushed
all the way to the right here.

And to achieve this,
we can go to this Row
that contains this Text and Icon button,
and there we can set the `mainAxisAlignment`,
which in a row controls the horizontal alignment,
and set this to `MainAxisAlignment.end`
to push the content to the end of the row like this.

We can also add
the `crossAxisAlignment`
and set this to `CrossAxisAlignment.center`
to center the content vertically.

Now, if you want to do that or not
is something that's up to you,
but you can do that as well.

With that, I do have the basic user interface
I wanna have here,
and we now of course have to make sure
that when we press this calendar icon button here,
we actually do open such a date picker
that allows the user to pick a date.

For that, we have to work on that function
that's passed as a value to `onPressed`
on this date icon button.

And I'll set this function up
as a separate method here in my state class.

It does not matter if you define it in front
or after this dispose method here.

Here, I'll define it in front of it,
and I'll name it `_presentDatePicker`,
because that is what we do in this method here.

We want to show that date picker.

Now, the other, it's this `_presentDatePicker` method

that should be used as a value for this IconButton here.

So this anonymous function can be replaced

with a pointer at this `_presentDatePicker` function here.

And inside of this function

or inside of this method to be precise,

we now wanna show such a date picker.

And this can be done

by using Flutter's `showDatePicker` method.

Now, before in the `expenses.dart` file,

you already saw `showModalBottomSheet`,

which was a built-in function.

Now, you see `showDatePicker`,

which is another built-in function supported by Flutter.

Just like `showModalBottomSheet`,

which wanted a context,

`showDatePicker` also wants a context.

But in addition, it also needs an `initialDate`,

so basically the date

that is selected initially when the picker is opened,

then the `firstDate` that can be selected,

so the lowest possible date, so to say,

and the `lastDate`,

so the maximum date that can be selected.

Now, to get all these dates,

I'll create a new helper variable

or a final helper variable to be precise,

which I'll call `now`, which is simply `DateTime.now`,

and this should be my `initialDate`.

So my `initialDate` is simply today,

created with `DateTime.now`

because `initialDate`, just like `firstDate` and `lastDate`,

all want `DateTime` values.

So therefore, here I created a `DateTime`
and used this as an `initialDate`.
For the `firstDate`,
I then want to create a new `DateTime`
by calling the `DateTime` constructor function like this,
and by now manually passing a year, month, and day to it.
And for the year, I want to use `now.year`.
So on this `DateTime` object,
which now is since we created it
with this `DateTime` constructor,
you can access various pieces of information
with help of all these properties.
For example, the year property
to get the year of the date here,
so of the now date,
so the current year, therefore,
and I wanna deduct 1 from it by using this `-` operator.
So this gives me the current year - 1.
Then the next value you have to set here
on the `DateTime` constructor function is the month.
And here I'll use `now.month`,
so the same month but one year ago.
And the last value I want to set is `now.day`,
so the day of this date here.
So basically, I'm setting the `firstDate` to today
minus 1 year.
And for the `lastDate`,
I will now simply use `now` again, so today,
so that my `initialDate`
and the last possible date I can set is now,
but the first possible date,
the oldest date that can be set is now minus one year.

And with that, if you save everything
and you force a reload,
you should be able to open this overlay
and click this date picker button here,
and now you get this date picker overlay
presented on the screen,
and here you can navigate between years
and the different months in a year
and then pick any date of your choice and confirm this.
And of course, at the moment we're not doing anything
with that confirmed value,
but that's how we can show the picker.
But of course, now we also wanna store that picked date.

19. Working with "Futures" for Handling Data from the Future

So, how can we now store
that date that was selected?
Well, `showDatePicker` in the end returns that picked date,
but wrapped into a so-called future,
a value of type `Future`,
which we haven't encountered thus far.
Now, futures are a very important data type built into Dart.
You will see futures again later in the course
when we work with HTTP requests,
and the future simply is an object from the future,
you could say.
It simply is an object that wraps a value
which you don't have yet,
but which you will have in the future.
Like here for `showDatePicker`,
when this line of code is executed by Flutter,
it opens this date picker and shows it on the screen.

But of course, no date has been picked yet.

That will only happen in the future, you could say.

Therefore, Flutter internally
basically memorizes that in the future
some value will be picked here
and it allows you to register a function
that should be executed in that future
once that value is available.

And that function can be registered
by calling a method on that object,
on that future object that is returned by `showDatePicker`.

So, this future object,
and that's important to understand,
is returned immediately,
but it doesn't contain the picked value yet.

Instead, this future object gives you a `then` method,
which you can call,
to which you can pass a function,
an anonymous function, for example,
which then will get that value,
which in this case here has been picked
once it has been picked.

So this function which you pass to `then`
will be executed by Flutter once the value is available,
so once a date has been picked here.

It's kind of similar to how buttons work,
where we also define a function
that will be executed in the future,
but here we're dealing with a widget
where we set this function
as a value for one of the many widget arguments.

In case of `showDatePicker`,

we instead get such a future object
to which we pass such a function,
but the idea is generally the same,
that we wanna get hold of some value
or some event in the future.

Now, using such a then method
is a one way of getting hold of that value.

A more convenient method
is to add the special `async` keyword
here between the parentheses after the method name,
before the opening curly brace,
and then add the `await` keyword in front of the code
that does yield such a future.

So you can't use `async await` here in the build method
for your widgets,
but you can use it
whenever you get such a future value.

If you use `await` here,
you get access to that `pickedDate` in this case here
as a value in a regular variable.

But even though this now looks like all the other code
or almost like all the other code,
the `await` keyword internally simply tells Flutter
that this value that should be stored in `pickedDate`
won't be available immediately,
but at least at some point in the future,
and Flutter should therefore basically wait for that value
before it stores it in that variable.

Nonetheless, Flutter will make sure
that the next line after this code here,
so this line here,
will only be executed once the value is available.

So it will really kind of wait for that value,
and therefore here we could print that pickedDate,
or we could store it in a variable,
possibly even with setState
so that we could output it here
in this Text next to this IconButton.
And that is indeed what I wanna do here.
So I'll add a new variable here,
_selectedDate,
which should be of type DateTime.
And since I am setting a type here,
the var keyword must be omitted.
And actually it initially will not hold a value,
hence it should be DateTime? to tell Flutter
that selectedDate either stores a value of type DateTime,
or nothing, null as it's called in Dart.
And instead of _presentDatePicker
here after waiting for the _selectedDate,
we can call setState
and set _selectedDate = pickedDate.
And pickedDate is of type DateTime,
so setting it like this is fine.
And with that, we can use this _selectedDate
to output it down there.
Of course, here I again want to output a version
that's human readable.
And for that reason,
I'll use the same approach I used before
in my expense model
where I use this intl package
and such a formatter
by adding it here

to this `new_expense.dart` file as well.

And this formatter can be set up outside of any class.

Alternatively, we could of course also simply import from our own package here.

So from our own project package

and there from the `expense.dart` file in the `models` folder

because in there we set up this formatter,

so it is actually available

as soon as we import from this file.

And we can use this formatter down there

to not output `_selectedDate` here,

but instead, I want to output `No date selected`

if we haven't selected a date yet,

so I want to check if `_selectedDate`

is equal to null,

and then we can use a ternary expression here

to output `No date selected` in that case.

Or alternatively, we can use the formatter here

and call `format` on the `_selectedDate`

to output a formatted version of that date,

and now we have to remove the `const` keyword here.

Now, I'm getting an error here

that in the end `DateTime` might be null here

and `format` wants a `DateTime` that can't be null,

there is no question mark after `DateTime` here.

Of course, we know that `_selectedDate`

won't be null here though

because we're checking this here,

and therefore we can add an exclamation mark at the end

to basically force Dart to assume that this won't be null.

So with an exclamation mark

after a possibly null value,

we tell Dart that this will never be null,
which we know with certainty here
because we're checking for null in quality here.
So with that, with all of that out of the way,
we can save this.
And if we go back to our application
and we pick a date here,
we can see that formatted picked date here.
And of course, we can also change that date
and it would update here.
So that's now working as it should,
and we can therefore now as a next step
work on that category selection dropdown.

20. Adding a Dropdown Button

So it's now really time
to finalize this format,
this overall overlay here.
For that, here in this last row
where I have these two buttons,
I also want to add a third button, a dropdown button
that can be used to select a category.
So therefore, here in this row next to the buttons,
I'll add this dropdown button widget,
which is offered by flutter, which renders a button
that will display a dropdown once clicked.
Now this dropdown button widget then has two arguments
that must be set.
The onChanged argument, the onChanged parameter as always
for these buttons, must be set to a function
that should be executed when the button is pressed.
Though here it really is a function that will get a value

of type dynamic

because the value

that it will get here will simply be the value

that has been selected from the dropdown.

And it's of type dynamic

because flutter, right now, does not know which kind

of items will display in this dropdown

because that's the next part.

Items now wants a list

of dropdown menu item values.

Now in this app here, I of course simply want

to display all these possible categories

in this dropdown menu.

And for that we can reach out to our category enum,

which is available because we're importing

this expense dart file in which this enum is defined.

And on that enum,

you can not just access the different values

that you defined, but actually also a values property.

So a property named values, which gives you a list

with all the values that make up the enum.

And that's what we need here

because we wanna display all these values in

this dropdown list.

Now, values is a list of enum values, a list of categories,

therefore, and items, once a list of dropdown menu items.

Therefore, we have to do something we did

before in the course already.

We have to call map on this list

to transform it from one type

of value into another type of value.

Now map, as you learn before in the course, wants a function

which will be executed automatically by dart
for every list item.

And we will get the list item
for which the function is currently executed
as a input value automatically passed in by dart.
So here it will be a single category
since this function will be invoked
for all the category items.

And we wanna return our dropdown menu item here.
So dropdown menu item, that's what we wanna return here.
Now, dropdown menu item in the end is a widget,
and it is a widget where we have to set the child parameter
to another widget, which simply defines
what will be shown on the screen.

And here, I'll keep this simple
and simply output a text widget here.
And the text value should be this category,
but category of course is of type category,
so our enum type here.

And to convert this to a string that can be output
with the text widget, we have
to access the special name property that's made available
by dart, along with a couple
of other properties on all those enum keys,
all those enum items.

And name, then, has a two string method, which we can call
to convert these enum type values here,
two strings that can be used with the text widget here.
So the values property on the enum itself was
a property provided by dart,
and the name property on the individual enum items is
another property provided by dart.

Now here, we're still getting an error in the end that we have an iterable here in a place where a list is expected, and therefore, we should do something we did before in the course as well.

We should call to list on the result of map.

But, this is actually not all I want to do here.

Instead on this dropdown menu item, I don't just wanna set the child, but in addition to that, also the value parameter.

And this will be the value that will be stored internally for every dropdown item.

It'll not be visible to the user, it's this child that's visible to the user, but it will be stored internally and it will be this value that we get here in this onChanged function whenever the user selects one of these dropdown items.

And here the value is the category itself.

So the specific enum item that is mapped to a specific dropdown menu item.

So therefore here in onChanged, flutter now understands that the value we will get here will actually be of type category.

It will be one of these enum items.

And then we can use this here.

And for the moment, I'll just print it.

Of course, in the future we'll do something else with it.

But with that, if we save everything and force a reload, if we open this, we have this dropdown button here, and in there, we find our different items.

Though, as you see, when I select them,

we actually don't see them here.

We also have no initially selected item,
and they're also all written in lowercase.

So there are a couple of things we have to do here.

And the first thing is to get rid of to string.

We actually wouldn't even need that

because name is already a string,

but I wanna replace it instead with to uppercase,

because this is a built-in method

that can be called on strings that transforms a string
to be all uppercase.

So that's the first step we should take.

In addition, I now wanna store the value that was selected

because just to quickly show you this, if you do go

to the debug console,

you can indeed see the different selected values there.

So printing worked and selected worked,

but I also wanna store the values now so

that we can later use them when our form is submitted.

And for that, we can't use a controller.

Dropdown button does not support a controller,

but we can of course manage this manually

by adding another new variable

to our new expense state clause just

as we added selected date there to manage it manually.

And here, the value will be of type category, of course,

and I'll name this selected category.

And initially I'll set it to category leisure.

So that's the selected category variable.

And now, of course, this should be updated

whenever a new value is selected,

so with help of onChange here.

Alternatively, we could also create a new separate method at which we point here, but here I'll do it just in place and call set state

to set selected category equal to value.

Now, value could be null.

It has this question mark here,

and that's the problem dart has with this code here

because maybe the user didn't select anything.

And therefore I'll add a little if check

and check if value is equal to null.

And if that's the case, I will simply return.

And we can do this outside of set state, actually, so

that this code here only executes if the value is not null.

Because if we return in a function,

no code thereafter will be executed.

A return statement is always the last code

that will be executed.

So therefore, if we return here, if the value is null,

this code won't be executed.

And on the other hand, if this code will be executed,

we know that the value will never be null,

and dart now also understands this and no longer complains.

So now that we're storing selected category

and we're updating the state whenever it changes,

we can also tweak this dropdown button

and actually add the value parameter to it

and set this to selected category.

And with that, we ensure

that the currently selected value will be shown

on the screen instead of showing this empty dropdown

as it was the case before.

With that, if you save that

and force a reload again, now you see
that leisure is selected initially.
And if I switch to a different category,
that is shown on the screen instead.
And you also see that it's all written uppercase
as mentioned before.
So that's looking really good now.
And therefore,
there are now just some tiny adjustments left.
For example, I want to add a spacer
between my dropdown button
and the other buttons, a const spacer, to make sure
that there is some spacing between all these buttons.
And I also wanna have some spacing to the top, so some space
between the amount input
and this dropdown, for example, some vertical space.
And for that, of course, we can go to this place here
before we start that row that contains the dropdown button,
and here add a sized box, a const sized box
with a height of 16, for example,
so that we have some distance here.
Now, with that, we made good progress.
We're almost done here, but not entirely yet, user.
I of course wanna make sure that if we try
to save some empty input fields, we fail,
and we show the user some error message that tells the user
that valid input values have to be entered first.
And of course thereafter, I also wanna make sure
that this list is updated
and the new expense is added to the list
before we then thereafter take care about styling
and the chart.

And last but not least, related to this form here

or to this overlay.

Of course, if you tap into one of these fields, you see

that the keyboard is coming very close

to these input fields,

and it might actually obscure these input fields

on some devices.

So we also wanna make sure that that will never be the case.

But we'll start with validation.

21. Combining Conditions with AND and OR Operators

So let's make sure we also validate

those user inputs when submitting them.

And for that, to achieve this, I'll first

of all add a new method here to NewExpense state.

And this could be named submitForm

or submitExpenseData, something like this.

And it should be triggered

whenever this button at the bottom here is pressed,

this one here.

Here, instead of printing some dummy values,

I want to trigger submitExpenseData by pointing at it.

And inside of submitExpenseData, we now have

to validate the data selected by the user.

For that we can use a if statement about which we learned

before in the course already

and check if titleController.text

after trimming it, which is a built-in method

that simply removes excess white space

at the beginning or end.

If this is then empty

and isEmpty is a built-in property

just as trim is a built-in method.

Trim can be called on strings and text is a string
and isEmpty can be called on lists and strings.

And the result of calling trim is a string,
which is why we can use isEmpty here.

So this overall will be true if we have an empty title
or just a bunch of blanks.

And if that's the case, then we wanna show an error message,
which we'll do soon.

But of course that's not the only case when we wanna show
an error message.

Instead, we also wanna show an error message if we have
an invalid amount.

Now the amount is not just invalid if it's empty,
but also if it, for example, is a text that doesn't qualify
as a number or if it's a number smaller than zero
because our amounts
for the expenses here must be greater than zero.

Therefore, as a first step, we have
to transform the amountController.text, which we get,
which is always a string, to a number.

So our enteredAmount as we could call it,
has to be a number, not a string.

And this transformation can be achieved
by using double.tryParse, a method built into Dart.

tryParse takes a string as an input
and then returns a double if it is able to convert
that string to a number, to a double
or it returns null if it's not able to convert it.

So using tryParse on a text like 'Hello'
Would yield null.

For example, tryParse on something like 1.12

as a string would yield 1.12

as a double, as a number.

So that's something we have to do here.

And with that done, we can add another variable,

which could be called `amountIsValid`,

which we wanna set equal to true or false, so to a Boolean.

But of course we don't wanna hard code true or false here.

Instead that should be derived dynamically

by taking a look at `enteredAmount`.

Just as we're taking a look at the title text here.

So what we wanna check here is whether `enteredAmount`

is maybe equal to null

because I just mentioned `tryParse` will yield null

if an invalid number was entered.

So some text that can't be converted to a number.

So now `amountIsValid` will be true

if `enteredAmount` is null

because indeed then the amount is invalid.

But that's not the only case in which we would consider it

invalid in this app here.

Instead, the amount will also be invalid

if it's smaller than zero as mentioned.

So here we actually wanna derive true or false

based on a combination of conditions.

And in Dart, you can do this

with two very important operators.

The logical and operator, which is written like this

with two ampersand symbols, or the logical

or operator which is written like this

with two pipe symbols.

And you will see this in many other programming

languages as well.

Of course thus far you already saw some operators like plus or minus, which are used in mathematical operations.

There also would be the asterisk operator for multiplication or the forward slash operator for division.

But where plus and minus

and so on are used to derive numbers,

and and or are used to derive Booleans.

And they help us derive Booleans

by combining multiple conditions.

Here, the first condition is that enteredAmount is equal

to null, in that case amountIsInvalid will be true,

or it should also be true if enteredAmount

is smaller or equal than zero.

So if one of these two conditions is met,

amountIsInvalid, will be true.

If both conditions are false, amountIsInvalid is false,

but it is enough that one of them is met

to set amountIsInvalid to true

because of this logical or operator.

Well, and with that, we now know when

amountIsInvalid will be true.

We can now add this here to our if condition

where we're checking whether the title is invalid

with this code.

Now we wanna combine this condition with amountIsInvalid

by again using the or operator, the or operator instead

of the and operator

because we wanna make it into this if block

where we wanna show an error message in the future

if at least one of our checks yields through.

Because even if all the input values are valid, it's enough

for one input value to be invalid to show an error message

because we need all input values to be valid,
not most of them.

So therefore here we want to now add `amountIsInvalid`.

And this is now of course not a condition,
but it is a value that contains true or false.

And that's all the if statement wants.

It wants a bunch of true or false values
or expressions that yield true
or false to know whether the code inside
of the if statement should be executed or not.

So we spent quite some time on this if statement already,
but it is important to understand how if statements work
and how these conditions work and how they can be combined.

And we're of course also still not done
because the date could of course also be invalid.

Therefore, we want to add another condition here again
with the or operator for the reasons mentioned before.

And here we wanna check if `selectedDate` is equal to null
because if that is null,
which it can be if no date has been selected,
then we also wanna show an error message.

We don't need to check the `selectedCategory`
because here we have a default category.

So here it should not be possible to set this to no value,
but for the other values it would be possible.

And that's why we're checking them here.

Well, and now we can make sure
that we show an error message in this if statement.

22. Validating User Input & Showing an Error Dialog

So how do we show an error message here?

Of course, it's important to keep in mind

that this message will only be shown

if we make it into this if statement,

which will only be the case if at least

one of these conditions here is met.

Now to show this error message,

we can use another built-in function provided by Flutter,

and this time it will be the showDialog function,

which will show some info or error dialogue on the screen,

some popup on the screen you could say.

ShowDialog, just like showDatePicker

or showModalBottomSheet before wants a context value,

which we set to this context object

that's available in this state class.

And it wants such a builder function,

which should be a function that takes some context

as an input and returns a widget as a value.

And that widget will simply be the content

of that pop-up dialogue that will be shown.

So here we get this context

provided by Flutter automatically

when it executes this builder method.

And the widget I wanna return here

will be one of the built-in dialogue widgets.

Now there are a couple of built-in dialogue widgets

and I'm interested in the alert dialogue,

which is a dialogue styled and optimized

for showing some alert message.

The alert dialogue widget then has a couple of parameters

that can be set.

For example, a title parameter, which wants another widget

where I'll use the text widget

to give this pop-up window a title.

For example, invalid input.

And this can be converted to a constant text.

I'll not convert the overall AlertDialog

because this would soon not be possible anyways.

Because the title is not all that's added here,

instead I also will add some content,

which is the main text,

the main content shown in the dialogue.

And here, I'll create another text widget,

though you could also use other widgets in here,

don't get me wrong,

but here I'll use some simple text and for example, say,

"Please make sure a valid title, amount, date,

and category was entered."

And now last but not least,

we can also set the actions parameter

along with many other parameters, as you can see, of course.

But here it's the actions parameter, which I wanna set,

which allows you to define a bunch of actions,

typically buttons that should be displayed in that dialogue,

which for example, allow the user to close the dialogue,

so to close this popup window.

And here I'll add a simple text button as an action

where onPressed, of course, must be set

to an anonymous function here where I then actually want

to close this dialogue by again using navigator.pop

as we did it before for the modal bottom sheet

and the pop method wants the context.

In this case, the context connected to this dialogue,

so this CTX value,

which is passed into this function here

automatically by Flutter.

And besides setting `onPressed`,
I of course also set my child here,
the main content of this button.
And here I'll set this to a text widget
where I simply say okay,
and turn this into a constant text.
And with that, we're showing this error dialogue.
Now, as a last step, I just also want to return
after showing this dialogue because as mentioned before,
whenever you return in a function,
you make sure that no code thereafter
in that function gets executed.
So here I'm returning after calling `showDialog`,
but still inside of this if statement,
so that any code I add thereafter
won't be executed if we made it into this if statement.
Because if we make it in there,
I just wanna show this dialogue and then do nothing else.
I don't want to continue because soon when we continue,
we will use the entered data and create a new expense.
And that, of course, should not happen
if invalid data has been entered.
So with all that out of the way,
if we save this and force a reload, if I open this
and I enter nothing, I get this error dialogue,
which I can close of course, but I get it
whenever I try to submit invalid data.
And that therefore, of course, works exactly as it should.
Now, as a next step, we can make sure
that if valid data was entered,
we use that data to update this list.
And this is actually something which you can try on your own

because this is actually something
which we basically already did in this course.
Of course not in this exact app, but the pattern,
the approach you need here is one we already used before,
but of course in the next lecture,
we'll also implement it together.

23. Saving New Expenses

So did you succeed adding
the expense that is created here
to this list of expenses?
Let's do it together now.
And for this we have to go to Expenses Dart
because that's where we manage this list of expenses.
And in here we can add a new method
that could be called addExpense
because that's in the end what we want to do.
And here in the end, I expect to get an expense value
that should be of type Expense.
So using our custom class here as a type.
And then here I wanna reach out to registeredExpenses
and call the built-in add method, which is
provided by Dart, on all lists,
to add a new item to the list.
And of course, here I want to add this expense as a item.
Of course, that's not all I wanna do though.
Instead this should be done inside of setState like this
to make sure that the UI is updated
so that the updated list is output here with help
of ExpensesList.
But that is all we have to do to add a NewExpense, though
of course, at the moment, addExpense is never executed.

Now it should be executed from

inside this `NewExpense` widget here.

And to do that, we have to go to that widget

and there add a new property, a new property

that will actually store a function as a value

as we already did it in our course sections

where you learned about this pattern of accepting functions

as the values in your own widgets.

And this should be a function that returns nothing

but actually provides an expense value.

And the property could be named `onAddExpense`.

And I choose this naming pattern to make it clear

that this in the end, will contain a function

that should be executed when a certain event occurs.

In this case, when this form is submitted.

Of course here,

we then also have to accept this argument,

this named parameter,

and I'll use that same name by using this this shortcut,

and I'll require it

because this widget won't work if we don't

provide this function.

So with that, we've got `onAddExpense` in our

`NewExpense` widget class.

But of course we needed inside

of this `NewExpenseState` class, we need it here inside

of `submitExpenseData` after this if check down here.

But as you also learned

before in this course, you can get access to properties

of the widget class that belongs to your state class

by using the special widget property provided by Flutter.

This is available in classes that extend the state class

as we're doing it here.

And this simply gives you access
to the connected widget class.

And therefore we can also get access
to this `onAddExpense` function property
and execute it as a function here because it is a function
and pass our expense to it.

And for this, we create a `NewExpense` here where we have
to set title, amount, and so on.

And the title of course is the value stored in
`_titleController.text`.

So this value here, that's what we assign here as a value
for title.

For the amount, we use this entered amount,
which I'm extracting here.

So this parsed text, which we validated.

So at this point down there, we know that it's valid,
otherwise we wouldn't have reached that code
because this return statement would've made sure
that we don't get there if one
of these values would've been invalid.

The date is the selected date, of course.

And here we get an error
because Dart thinks that it could be null,
but we know that it won't be null down there thanks
to our validation.

Hence, we can add an exclamation mark to tell Dart
that we're sure that this won't be null.

And the category here is of course, the `selectedCategory`.

And with that we're creating our expense
and we're passing that expense as a value to `onAddExpense`
and therefore indirectly to `addExpense`.

And now we just need to make sure that this `addExpense` function is passed as a value to `NewExpense`.

So therefore here we simply have to refer to `addExpense`.

Now with that, we just have to remove `const` here.

And this of course is also not a positional argument, but instead the value that should be set for `onAddExpense`.

So this named `onAddExpense` parameter.

And with that, we're passing this `addExpense` function as a value to `NewExpense` so that it can be executed from inside there.

And if we now save that all, and we try this and enter some values here and select a date, and then also pick a value here and save this, we can close this.

We'll have to make sure that it closes automatically soon, but we can close it manually for now.

And we see our `NewExpense` here with all the data we entered in our input fields.

Now with that, we're able to submit our data after validating it and we can add a new expense.

But of course, now as a next step, I wanna make sure that this overlay actually closes after we submitted the valid data.

And I wanna make sure that this keyboard, which opens up as we tap into these input fields, doesn't possibly obscure any buttons or anything like that.

So that's what we'll work on next.

24. Creating a Fullscreen Modal

Now let's start by making sure that we actually close this overlay

after a new expense was added.

For that here in `new_expense.dart`,

after calling `onAddExpense`,

inside of this `_submitExpenseData` method still,

we can use the good old Navigator class,

which we saw a couple of times already,

and call the `pop` method, and pass the context to it,

this context property that's managed

by Flutter in this state object here,

to make sure that the overlay is closed,

because of course we wanna do that

after adding a new expense.

Now to make sure that this keyboard

that opens up doesn't overlap our inputs

we have to go to the place where we open

this modal bottom sheet,

and there is set one new parameter, `isScrollControlled`,

and set this to true.

When you set this to true,

the modal overlay will actually take the full available

height and therefore ensure that, of course, now

we don't overlap any inputs.

Of course, you can see, though, that at the moment

this also means that some content is obscured

by some device features like the camera here.

And to make sure that this is not the case,

we can go back to the `new_expense` dart file.

And here where we add some padding around our form,

we can switch from `all` to `fromLTRB`, which simply allows us

to set different spacings for different directions.

left top, right, and bottom.

And set all these values to 16 except for the second one,

which is the one to the top, as you can see here.

And we could set this to 48,

which should be enough no matter on which device

you are working here.

With that things look good here.

So this is now actually how we can make sure

that we never obscure anything here.

25. Using the Dismissible Widget for Dismissing List Items

So, this isn't looking too bad now.

We can now add new expenses here with amount, date,

category and a title.

Those expenses are then added to the list.

Of course, the styling as mentioned, isn't finalized yet.

We'll take care about that soon.

But before we do that,

I wanna make sure that we can also get rid of expenses

by swiping them away.

So by swiping from right to left.

And at the moment, this is not possible,

but it's such a common pattern that Flutter also helps a lot

with implementing this pattern.

So, how can we make sure that these items

can be swiped away?

Well, for that, we can go to our `expenses_list`

where we have this `ListView.builder` with our expense items.

And Flutter gives us a special widget,

which we can wrap around list items that should be swipeable

or that should be dismissible, removable to be precise.

And the widget conveniently is named `Dismissible`, like this.

Now `Dismissible` wants a child,

and that should be the list item that can be swiped away.

So in my case, my `ExpenseItem`, but it then also needs a key.

And I mentioned this key before,

it was always this thing which we accepted here

in our constructors and forwarded to the parent widget.

And that was all we had to do with keys thus far.

Now the good news is that in most cases, in most widgets,

that is all you have to do with keys.

There is a good reason why we haven't seen keys

being used anywhere besides registering them here

in our constructors.

But there sometimes are scenarios

where those keys are actually needed

because this key mechanism exists to make widgets

uniquely identifiable.

And I'll dive a bit deeper into all of that

in a later section.

But in the end here,

where we should be able to swipe widgets away

and then remove the data that's associated with a widget,

a key is needed to allow Flutter

to uniquely identify a widget and the data

that's associated with it.

And such a key can be created by using the built-in

`ValueKey` constructor function.

This creates a key object, which can be used as a value

for this key parameter.

And as mentioned, for most widgets,

you don't need to set any keys,

you don't need to worry about keys at all.

But here for the `Dismissible` widget,

a key is needed to make sure that the correct data

will be deleted.

So here we must create such a key,
and ValueKey wants an input,
in the end it wants some input that can be used
as a unique identification criteria
or value for the widget to which the key is attached.
And in this case here,
we could simply take our expense identified by index
as a value for ValueKey,
and that's all we have to do regarding that key.
With that, if we add a comma for formatting and save this,
and maybe force a reload,
you will see that you can now swipe those list items
and this then removes them.
Now the connected data is actually not removed,
therefore, if I add a new item here real quick,
a new expense, you will see that I get an error
basically because we have a mismatch
between what's showing on the screen
and which data is still stored internally.
But this is generally how you get this swiping effect.
Now of course, we must make sure that we also do something
when we swiped a widget away.
And for that, you can add the onDismissed parameter,
which allows you to trigger a function
whenever one widget has been swiped away,
so when it has been dismissed.
And here I of course wanna trigger a function
which then removes the associated data
from my list of expenses.
So I wanna update the registeredExpenses list
whenever I swiped one of my expense items away.
Therefore, we should go to the expenses dart file

to the `ExpensesState`, therefore,
where we also added `addExpense` before
and add another method here.
And this other method could be called `removeExpense`
because that is what I want to do here.
And in the end here, I want to get the expense
that should be removed, and we can then call `setState` here
and reach out to the `registeredExpenses`.
And there, call `remove` and pass the expense
that should be removed as a value.
And that should make sure that the expense
is not just removed visually from the screen,
but also internally from that list of expenses
that we managed.
Now in `expenses_list`, we must accept this function
as an input so that we can call this `removeExpense` function
from inside this widget.
Therefore, as we did it multiple times before
in this course already, we can add a new property
which actually wants a function that returns nothing,
but that takes an expense as an input,
and that could be called `onRemoveExpense` here.
And of course, this must be set
when the `expenses_list` widget is created.
So, we require it and then bind the named
`onRemoveExpense` parameter to this property.
And therefore, of course, down here,
we can execute this function.
However, we can't point at it like this
because `onDismissed` actually wants a function as a value
that takes a so-called `DismissDirection` as an input,
which simply is some information that tells us

whether we swiped from left to right or right to left,
so that you could in theory do different things
depending on how the user swiped.
Now here all simply set us to an anonymous function,
therefore where I get this direction
and I'm not really interested in the direction,
but I still have to accept it here.
And then I can call `onRemoveExpense`
and then pass my expense as a value to `onRemoveExpense`,
and basically ignore this direction.
Well, and with that, we just have to make sure
that `removeExpense` is also passed as a value
to `ExpensesList`.
So therefore here, we now have to set this newly added
`onRemoveExpense` named parameter and set it to a pointer
that points at `removeExpense`, like this.
And with that, if you save that and force a reload,
if I now swipe this away, it's still gone,
but now you will see that as I add a new expense here,
I don't get that error because now the UI is in line
with the internally managed data.
And that's how easy it is to make items
in a list dismissible.

26. Showing & Managing "Snackbars"

So we can now also remove items.
There are two things related to that
that I wanna add to this app now.
For one, it would be nice to have some fallback message here
if we have no expenses at all.
In addition,
I wanna show a little info message at the bottom

whenever we did remove an item,
which also offers us a button,
which we can click to bring back that expense item,
in case we swiped accidentally.

Now, let's start with the easier thing
with showing that fallback message,
instead of showing that list, which would be empty.

To do that, we can use a ternary expression,
or to mix things up, add a new variable here,
which stores a Widget,
which we could name `mainContent` or anything like this.

And, initially, here, we could use the `Center` widget
to show some centered text
where we say "No expenses found. Start adding some!"

Something like this.

And this can be a `const` value here.

And we wanna replace that if we do have expenses.

So here we can add a `if` check
and check if `registeredExpenses` is `NotEmpty`,
another utility property offered by `Dart`
which we can use on lists
to find out whether a list is not empty.

And if `registeredExpenses` is not empty,
we wanna overwrite `mainContent`
and store a different value in that variable instead.

And that different value should be this expenses list here.

So I will grab that list and cut it,
and store that in `mainContent` instead.

And now it's of course `mainContent`
that should be output here as a child in this column.

Therefore, here, I see "No expenses found,"
but if I add a new expense here, like 16.99,

and maybe this is TRAVEL,

we see that instead.

So this works.

But now let's work on that message that should be shown

if we dismiss an expense.

To show a message after deleting an expense,

it sounds like it would be a good idea

to go to this `removeExpense` method,

because that is where we do remove expenses.

And indeed it is here after calling `remove`,

where I do want to show this info message.

And we can show such a message

by using another feature offered by Flutter.

Besides all these utilities show functions

which Flutter offers here,

it also offers a special object,

which you can use for showing or hiding certain UI elements.

And here, that would be

the `ScaffoldMessenger` utility object which we want to use.

This `ScaffoldMessenger` object has a `showMessage` method,

which we must execute,

which wants a context value,

which as you learned, is available in this class,

which is based on the `State` class.

Now this then gives us an instance

of this `ScaffoldMessenger`.

And this `ScaffoldMessenger` thing

offers us various utility functions and methods

that help us with controlling the user interface,

and showing certain things on that interface.

Most importantly,

we can call `showSnackBar` to show a so-called `snackBar`,

which you might have not heard before,
but which in the end is just a info message,
which is shown on the screen.

And `showSnackBar` wants a `SnackBar` value.

And this value can be created
by using the built in `SnackBar` class,
and instantiating this class here.

And `SnackBar` then needs at least a content
which must be set to a `Widget`.

And in most cases, this will probably be some plain text.

So here, for example,
we could say "Expense Deleted,"
because that is what we did,
and this can be set to be `const`.

But there is more that can be configured
about that `SnackBar`.

For example, it also has a duration parameter
that can be set and that wants a duration value,
a value of Type: `Duration`,
which can be created with the built-in
`Duration` constructor function,
so the built-in duration class.

And this then has various parameters that can be set
to define a duration.

For example, we can set it to 3 seconds here.

And this can also be turned into a constant value.

And we can also add an action to our `SnackBar`.

So not actions, but one single action,
and this must be of type `SnackBarAction`,
which is created by using
the special `SnackBarAction` constructor,
so the `SnackBarAction` class.

And this then has also some configuration items,
some parameters we can set,
like for example, the label, which does not want a widget,
but a plain string.

For example, Undo,
so that we have a button on that info message
that allows us to bring back that deleted expense.

And we then also have to add the onPressed parameter
to define the function that should be executed
if this Undo button is pressed on this SnackBar.

And in here, I then want to call setState to update the UI,
and change registeredExpenses,
and instead of removing something,
here, I wanna add something again.

However, actually not with the add method,
but instead with the insert method,
which also adds an item to a list,
but we add our item to a list at a specific position,
because I of course wanna bring back that expense
that we removed in exactly the same place
where it was before.

Now for that, we first of all need to find out
where this expense was
by getting the expenseIndex,
because insert wants the index
where the element should be added,
and then the element that should be added.

So we first of all need to find out which index
this expense that we removed had,
and for that, of course, before removing it,
we need to get that index.

We can get this by reaching out to registeredExpenses,

and then, there, we have a `indexOf` method provided by Dart to which we pass a value.

And, as a result, we'll get the position of that value, the index of that value in that list.

So it's then this `expenseIndex` which is passed as a first value to `insert`.

And the second value is that expense, which we deleted before.

It was removed from the list, we can still access it down here though, and it will then be brought back into the list.

And that's how we can display such a info message with such a `Undo` button.

If you now save that and force a reload to bring back those dummy items, you see that if I swipe them away, I get this info message at the bottom, and it disappears after three seconds.

But I can also press the `Undo` button to bring back the item.

And if I force a reload again, so that I have both items back, you will also see that if I remove `Flutter Course`, which is the first item here, and I bring back that item, it's brought back in first place, because that index was used to insert it in the right place.

I can also remove both items.

And, in that case, we might want to make sure that the info message that belonged to the item that was just removed is immediately cleared, so it's also removed from the screen, and we show the next info message instead.

Because at the moment, as you can see,
if I remove both items,
we get the first info message,
and only after this one disappears, the next one appears.
This might not be what we want,
and we can make sure that we clear
all existing info messages that we might have on the screen
by using `ScaffoldMessenger.of(context)`.
And there, we can call `clearSnackBars`,
which removes any snack bars,
so any info messages we might still have on the screen.
Now with that if we saved it, if I remove both,
you see the old message is immediately removed,
and the new one is shown as I remove multiple items.
So that's how we can show those info messages,
and how we can bring back items
if we accidentally removed them.

27. Getting Started with Theming

Now at this point, from a feature point of view,
this app is almost done.
Sure, the chart is still missing,
but we can manage our expenses here.
We can add expenses and delete expenses.
We can undo such a deletion process if we want to.
So that's looking good.
And of course we'll get back to the chart.
But before we do that, it's finally time to come back
to styling this app
and to take a closer look at Flutter's theming mechanism,
which you may use for styling your apps.
Because of course, whilst this app doesn't look super ugly,

it's also not really that appealing visually.

And of course, what we could do

to make it more appealing is we could style the individual parts manually just

as we did it in other sections in this course.

We could, for example, set up a linear gradient

for the background of our scaffold widget

as we did it in this quiz app we built

earlier in the course.

And we could use techniques like this

to style this app here,

this expense tracker app, if we wanted to.

And of course, there's nothing wrong with this approach,

but it would also be nice if we had one central place in

which we could set up certain styles

and those would then automatically get applied

to certain widgets

or could at least easily be referenced from

inside our different widgets so that we, for example,

don't have to copy

and paste color definitions across

different parts of our app.

And this can be achieved with Flutter's theming system,

which we can set up here in main.dart in our material app

widget because this material app widget,

which is this root entry point to our app, so to say,

this root widget

that contains all our other widgets in the end.

This material app widget is the widget you use

for setting up your app's theme.

Because I mentioned before that this app widget does a lot of behind the things stuff,

but it also gives you some key configuration settings that you can make here that influence your entire app. And the theme is one of those settings. And indeed, at this point here, I already have a theme set up. I got theme data set to use material three to make sure that my app here uses this material three look. Now, this might not be necessary anymore at the point of time you are viewing this video because material three might be the default by then, but this theme parameter that can be set on this material app will be useful nonetheless, because setting up a theme is not just about switching between the material three or material two looks, but instead you can use this theming mechanism to control in detail which color different widgets in your app will have, and which colors you want to be able to use in your app, and which text styles you want to be able to use and so on. So how can you then customize this theme? Customizing your applications theme involves setting this theme parameter here on the material app as we're already doing it. And theme wants a theme data object, which can be created with this theme data class provided by Flutter. Now, when instantiating this theme data class, you get many parameters that you can set to fine tune your app's theme. And this can be super overwhelming. And I will also say that Flutter's theming system, whilst being quite powerful and flexible,

also evolved a lot over time

and therefore this theme data class here also allows you to set a lot of parameters which still work but shouldn't be used anymore because they might be removed in the future and there are better ways of configuring your theme.

It's also worth pointing out that if you create a theme data object like this by instantiating theme data and then passing some configuration between those parentheses, you are actually telling Flutter that you are setting up the entire theme from scratch, which means that theoretically you really should configure all these aspects that make up your application. You should configure all text styles you could possibly use. You should configure all other styles that might be used by any widgets in your app. And whilst this of course can make sense for certain apps, it also means a lot of extra work because you might be happy with some default theme styles that Flutter applies otherwise.

Now of course, you could argue that this app here by default doesn't look that horrible, and that is correct, but you'll see a difference if I switch to a different way of setting up my theme - a way which I would actually recommend using. Instead of creating your theme data like this.

It's better if you don't pass anything to this constructor here and you instead call copy with and then to copy with you pass your settings. Like for example, in my case here, use material three true.

If you do that
and you force a reload, at least for me, I always have
to force a reload
to make these theme settings have an effect
on the running app.

If you do that, you will see
that it now looks slightly different.
It looks slightly different.

For example, this text in the title here looks different,
and I would argue looks a bit better
because now we're actually preserving some better default
styles that are applied by default, by Flutter,
and we can now override selected styles with
that copy with method.

So instead of setting up an entire theme from scratch
and having to style everything from scratch,
we're now using some default themes set up by Flutter
and we're only overriding selected styles
or selected aspects of that theme, you could say.

For example, we could now also target another parameter
here in this copy with method, which we're calling,
and set the scaffold background color to any color
of our choice, and maybe not red, I'll just use this
as a starting point to pick a color.

But then we could go for something like this,
like a light purple-ish color.

We'll change this later anyways,
but we could do that force those changes to be applied.

And now you would see this background.

Of course, this doesn't look great yet,
but this is how you can now overwrite individual parts
of the overall theme.

And here you can see that it's now in one central place where I can make certain adjustments like this scaffold background color, and it will then automatically apply to all the widgets that are interested in that setting.

In this case, the scaffold widget.

And if I would have an app with multiple screens, so with multiple instances of the scaffold widget, all those screens would use this color.

So that's how theming generally works and how you generally, in most cases, wanna apply your changes to a theme.

Instead of setting up an entire theme from scratch, it's often better to copy an existing theme and then just style individual aspects of that theme.

28. Setting & Using a Color Scheme

Okay, so now that we have a first idea of how the theming system generally works and how we might wanna adjust the theme, the question of course is how should we now adjust the theme and which parameters should you set here?

Well, in general, as always, it is a good idea to simply play around with different parameters here, but there is at least one thing you should know before you start playing around with that.

When it comes to setting up colors, there are many different parameters you can target here, and you have to understand that this `copyWith` method allows you to set up some base colors, like `accentColor`, or `primaryColor`, which will then be used by multiple widgets that are built into Flutter.

And you can also use these colors in your own widgets,
as I will show you later.

But besides setting some general colors like this,
you can also set individual colors,
like the `scaffoldBackgroundColor`,
or you can set up entire sub themes, you could say,
like the `elevatedButtonTheme`,
which requires a separate `ThemeData` object,
which then allows you to set up some button specific styles.

And this can of course all be overwhelming,
which is why we will set up these things step by step.

But when it comes to setting up colors,
there is a certain approach which I recommend using,
and that is the usage of a `colorScheme`.

And that's what I meant before,
the theming system in Flutter
saw a lot of changes over the years.

And `colorScheme` is now not a new approach,
but arguably the easiest approach
of applying some colors to many widgets
without having to manually
set a bunch of colors individually.

Instead with `colorScheme`,
the idea is that you define one color scheme
and Flutter then infers colors,
different shades of colors to be precise,
for different widgets.

And for that, I'll first of all create such a color scheme,
and I'll do that outside
of this `ThemeData` constructor function
to keep my code a bit more organized and structured.

So here I'll create a new variable

and I'll name it `KColorScheme`.

And starting with `K` is just a convention

you might wanna use in Flutter

that you set up global variables

with the starting character `K`.

It's not something you have to do,

but it is something you'll see quite a lot,

especially for theme-related variables in other projects.

So that's why I'll use it here as well.

And the idea here simply is that this variable

now contains a color scheme which we can use down there

for setting up our theme.

And a color scheme can be created

by using Flutter's `ColorScheme` class.

And whilst you could also create that from scratch,

which again would require you

to set a lot of colors manually,

`ColorScheme` also gives you a `fromSeed` method

or a `fromSeed` constructor function, which I recommend using,

because that is actually the constructor function

that will then automatically create a color scheme

with different shades of colors based on one base color,

one `seedColor` that you set up.

And here I'll create a color manually with `Color.fromARGB`.

And here I picked a color in advance

where my alpha channel has a value of 255,

red has 96, the green channel has 59,

and blue has a value of 181.

And this is a constant value, so we can add `const` here.

And with that `ColorScheme` defined here,

we can use that as a value down there in our theme.

And if we do that, and again, we enforce those changes,

now you don't see a lot of changes here,
but you see them, if I for example, open that modal.
Here, you see the buttons now have this purple color.
And the background also has a light purple shade at least.
Same is true for these cards here,
which have a light purple background.
And of course, we're also not done yet,
because now with that ColorScheme set up,
you can fine-tune your theme.
Besides setting up the ColorScheme
when setting up the theme,
you can now also overwrite selected parts of the theme
based on your ColorScheme though.
For example, you can set up an appBarTheme here
and create an appBarTheme object.
Again, best done by creating an object
and then using copyWith,
so that you get some default AppBarTheme settings
by Flutter, and you then just override selected settings.
And here you can then, for example,
override the background color of the app bar
and set this to a color from your ColorScheme.
So reference your ColorScheme here,
but now you get different properties on that scheme object
that were inferred and set automatically by Flutter
based on your seed color.
For example, here, I'll choose onPrimaryContainer
as a background color for my app bar.
And I can only encourage you to try out different colors
that were inferred here to see the difference.
But I'll go for onPrimaryContainer.
And I'll also set a foreground color for my AppBarTheme

and also set that based on my `ColorScheme`.

And here I'll choose the `primaryContainer`,
so that I pick these two colors from my `ColorScheme`
that is based on this seed color
to fine-tune my `AppBarTheme`.

And there are more things you could set
on that `AppBarTheme`,
for example, that the title should always be centered,
or that there should be some drop shadow behind it.

These are all things you can set,
and again, definitely play around with these settings,
but since we're using `copyWith`,
we don't have to set these settings.

Instead we're using the default settings
and we're just overriding `backgroundColor`
and `foregroundColor`.

And here we can then actually also use `const`.

And with that, if we now save this,
you see that I have a totally different look
for my app bar here
because of these selective overrides
which I made in my theme object.

29. Setting Text Themes

So creating a color scheme
and using that as a starting point for your copied theme
and then overriding some selected
aspects of your theme is the way to go.
Now, there are of course, more things that you can override.
For example, we could also apply a card theme,
which will affect all card widgets.
And in case you forgot,

the card widget is the widget we're using
for our expense item here.

So for these items in our list of expenses here,
they use this built-in card widget
and we can override the default styling of the card widget,
which is based on our color scheme,
but which might still not be the styling we want
with help of this card theme.

This wants a card theme object
just as app bar theme wants an app bar theme object.

And just as with the app bar here, I will also copy this.

And that's basically always the approach
you wanna use with theming,
you want to copy an existing theme.

For the card theme here, we can, for example,
target the color parameter alongside other parameters
that you could use to fine tune the look of your cards.

And we could set the color here to our color scheme again,
and now here select the secondary container, for example,
which is another color that was inferred by Flutter.

In addition, we could set the margin
to an edge insets object
just as you do it if you configure it
right on the widget when you use it.

But here you could set up a margin
for all the cards used anywhere in your app
if you had a certain default margin
that you always wanna apply.

And you could, for example here, set a base margin
that will always be used for all cards of 16.

And now here we can also use const and also use const here.

And again, save this and enforce this.

And now the cards would look like this.

And this is actually not bad,

but it's a little bit too much spacing

between those items for me.

Hence, I will actually switch from edge insets all here

to edge insets symmetric,

and keep the 16 pixels horizontally,

but vertically I'll switch to eight

to have a little bit less spacing vertically

between my different cards.

Yeah, I think that looks better.

Now I also showed you before that you can, for example,

set up some default button styles

by targeting the elevated button theme.

And here we want to create an elevated

button theme data object.

And on this object right now, at least

there is no copy with method.

Instead here we always have to create a new object,

but this object takes only one parameter,

the style parameter, which wants a button style object.

And here we could create such a style object

manually with button style,

but then we would again create it from the ground up,

which can sometimes be what you want

to have full control over how the button looks.

But often the better approach

is to use elevated button style

from which again uses Flutter's default styling

and then allows you to override some selected styles.

Unfortunately, we have inconsistencies like this

in the theming system where sometimes it's copy with

and sometimes it's an approach like this,
but that's why I'm showing this to you
for some of the most important widgets you'll work with.
And as always, the official documentation
and also simply Google is your friend
when it comes to finding out
how a specific widget is best themed.
Here for buttons,
I recommend using style from
on the elevated button in this case.
And now we can again target some individual style settings
for all the elevated buttons in the entire app
because that's always the idea behind theming,
you always set up styles for all widgets in the app.
And here we could, for example, change the background color
of our elevated buttons
and set this to K color scheme,
primary container in my case here.
And if I do that and we enforce these changes,
you see that now my elevated button here
has a more visible background color, not super prominent,
but a bit more visible,
which to me looks better than the
almost white background color it had before.
Now I'm almost done with setting up the theme
I wanna use for this app here,
but there's one last setting I also wanna apply here
before we then take a look at some other
theme related topics.
And the last parameter I want to target here
in my general main theme data object
is the text theme parameter.

Because of course in our application,
we also have many places where we output text,
like for example here in the title.
And of course the default text styles might work for you,
but you can of course also fine tune those.
And the best way of fine tuning them on a global level
is to set the text theme parameter
in your theme data object.
Now this one's a text theme object,
and of course as always,
you could create one from the ground up
and then target the different kinds of text
you might be using in your app manually here.
But I personally prefer creating a theme data object
here again, and then on that object,
accessing the text theme.
And then on that text theme here,
you can again call copy with.
This ensures that you're overriding the text theme
that belongs to your theme data object here.
Now, in copy with,
you can then target different parameters.
And these parameters represent different kinds of texts
that are used in different parts of your application.
For example, the text used here in the app bar
is the title large text.
Now how do I know that?
Well, the official documentation of course helps with that.
But in addition to that,
you can also simply try out setting those
different text styles and see which texts
change in your application.

Title large as mentioned,
should allow us to change the text style
of all the places where main titles are presented
like in this app bar.

And now to change the text style,
you have to set a text style object
as a value for title large.

And of course here you can create one from scratch.

Alternatively, you could also access
`themedata.texttheme.titlelarge`, and then use copy with
to just override selected parts of the text style
of your main title.

It depends on how much you wanna change.

Here I'll create a brand new text style,
and then we could, for example, change the font weight
and change it to font weight normal if we wanted to.

We can of course also set the text color.

And here I'll use my color scheme
and then use on secondary color
as a default color here for me
and maybe set the font size to 14.

If we do all that and enforce those changes,
you see the title now got smaller and is not bold anymore.

Now, regarding the color, you of course have to be careful
because we're also setting a foreground color
in our app bar theme,
which typically should override any text color you set here.

You can prove that if you set it to a color like red,
which would be clearly visible.

Even though I set it to red here, it's not red here
because I have that separate foreground color
in my app bar theme.

But of course, other places where I use this title large in my application might have a red text color. So here, I'll switch back to the other text color, which I actually wanna use, but it is important to understand how that works. And with that, I have set up a couple of styles and a couple of theme settings that are automatically picked up by various widgets in this app, as you can tell. And again, definitely feel free to play around with all these settings to get a better feeling for how you can influence the general look of your application. But this is also not all you can do with theming. You cannot just change the general look of your application or of the built-in widgets. Instead, you can also reference settings you made here in your theme data setup from inside your custom widgets to pick up certain styles and override styles in your own widgets based on your general style setup. And that's what we'll do next.

30. Using Theme Data in Widgets

So we got our theme set up. And this is of course very useful because it means that we don't have to manually override all these stylings in different parts of our app. But sometimes you of course wanna override certain styles in your app, but you also at the same time might want to use, for example, colors from your main theme.

And how could you do that then?

Well, let's, for example, start in expense item.

Here, in expense item, we might want to change the look of our expense title, and for that of course we can add the style parameter to this text widget as we did it before in this course as well.

And we could manually create a text style object here to set up the styling of the text.

But now that we learned about theming, we can also reach out to our theme stylings and apply those to this text here.

And to do that, we can use the theme class, built into Flutter, which has a `of` method which needs the context as a value.

And that is a pattern we saw before with the scaffold messenger.

There, we also had such an `of` method to basically reach out to this metadata context object managed by Flutter.

The theme class needs to do the same thing.

It is constructed by connecting it to this context object and this connection is needed

because through that context object Flutter gets access to our theme settings, basically.

And now on that theme class or on that theme object that was created here, we can for example,

access the text theme property to get access

to all the text themes that we set up

and all the default text themes

that were set up automatically that we didn't overwrite.

And then, for example, reach out to title large.

And this will then apply the text styles that are set up for title large.

So either the default text styles

or as in this case our own textile,
which we set up for title large.
And this text style will now be applied
to this expense title.
And you can see this more clearly if we go back
to the main dart file
and we changed the font weight
of our title large here from normal to bold
and maybe increase the font size to 16.
If we do that and force a reload, you now see
that this title here in our expense item
clearly is bold and has a different look.
Please also note that it has a different color than we have
here in the app bar, because of course,
as mentioned before, in the app bar,
we're setting a separate foreground color, whereas here
we're setting a different general color for title large.
And it's this color here, which is now picked up
by expense item.
As you can again see if I change this back
to colors red, briefly of course only.
But if I do that, you see that now the title is indeed red.
So that's how you can use some settings you set up
in your general theme, in your own widgets.
And of course here I'm switching back
to the actual color I want to use.
So that's how we can pick up theme settings.
Now of course here, I also wanna make sure
that this title is actually pushed all the
way to the left here.
And to achieve this, I don't need to do any theming.
Instead, I can simply go to this column

that contains my text here

and set the cross axis alignment,

which for a column is the horizontal axis

to cross access alignment start.

And this will make sure that all the column items are moved

to the start of the column horizontally,

so to the left in this case.

So that's how we can reach out to our theme

to use certain values from that theme.

But we're not limited to using text settings, for example,

in my expenses list, I also want to add a background color

to this dismissible item so that as we swipe an item away,

we actually have a little background effect that indicates

to us that the item will be deleted.

And this can be achieved by going to the dismissible widget

and setting the background property or parameter there.

Now the background parameter wants a widget,

and for that here we could use the built-in container widget

because this widget allows us to set a lot

of different parameters, including the color,

which is actually the background color of the container.

As you see here, the color it paints

behind a possible child widget you might

include in the container.

And that therefore allows us to set up a background color.

And this of course wants a color.

Now, as before in this course,

we could create a color manually here,

but now that we learned about theming,

we can also pick a color with theme of, from our theme.

And that allows us to pick a color that's more in line

with our general color scheme.

And to do that, we can access theme of
and then .colorScheme to get hold of
that color scheme we set up in our main dart file.
Remember, we created this color scheme with one seed color.
Now as mentioned, this behind the scenes creates a lot
of different shades and variations of that color
for different aspects of the app.
And we can now use those shades
and variance here to access the error color,
which is one of the automatically generated variations
of our seed color for error scenarios.
If I do this and I save everything,
you will see that as I swipe this item here,
I get this red background container coming in,
indicating to me that this is going to be deleted.
Of course here, we then also could apply some fine tuning
and for example, add a margin with edge insets
and then use symmetric to add some horizontal margin
of 16 so that with that, if I force a reload,
this container actually starts where the
list item also starts.
But of course here, we would have the problem
that if we ever change the margin of the list item,
we have to change it here as well.
And you might remember that in main dart I actually set the
margin of the card theme here to horizontal 16.
So that's actually where this margin here is coming from.
So therefore, since this is defined in our theme
in this application here, we of course
can get this margin from our theme, by again
using theme of and then getting the card theme
and there the margin, or if I only wanna apply

the horizontal margin as it's the case here,
I still create an edge insets object
with symmetric to set my horizontal margin.
But this value is then derived from my theme
by accessing `cardTheme.margin.horizontal`.
And here I'm getting an error
that in the end margin could theoretically be null
because I might not have set it up in my theme,
but here I know that it was set up.
Hence, I can add an exclamation mark
to get rid of this error.
And with that, I get the same effect as before,
but now it's coming from my actual theme.
And these are all things you can do.
Now, being able to use theme data
for fine tuning certain widgets is a very useful feature,
and it does not stop at simply using values
defined in the theme.
Instead, you can also extend or change those values.
For example, for a color like error
or any other color, you can also use one
of the built-in methods Flutter offers, on colors like
with opacity to make this color slightly transparent,
for example, by adding an opacity 0.75.
If you do that, you will see that
now this red tone is slightly transparent.
Now you might or might not want this,
but it is important to understand
that you can adjust those theme controlled values
in the places where you're using them.
For example, in expense item, we're also not limited
to using the title theme like this.

We could also use copy with here to use our general title large theme settings, but then overwrite them with specific settings that we want to use specifically here in this widget. So, which will not be changed in the general theme, but which will be adjusted in this widget whilst still basing our general style on the theme settings. Now here, I actually got no adjustments I wanna make, but it is important to be aware of these fine tuning possibilities.

31. Adding Dark Mode

So at this point we had a quite detailed look at theme-ing. There's just one last theme related topic I also wanna discuss here, and that would be how you can apply a dark theme or how you can allow the user to switch between a dark and a light theme. Because at the moment we have this light theme with a lot of white in it, and some users might prefer dark mode. Some users might even set dark mode as their intended default in the device settings. Like for example, here on Android where I can search for dark and turn on dark theme. Now my Android phone is in dark mode, but of course my app doesn't care. That's why you might want to provide an alternative dark theme. And for that, to do that we could create a second color scheme,

the dark color scheme.

And here again, we can do this with color scheme fromSeed, but the seedColor I wanna use here for my dark mode is now a color, which I also picked in advance, which uses the fromARGB constructor.

And then values of 255, 5, 99, 125.

Of course, again, you can play around with different colors.

That's just one color I picked here.

And with that dark color scheme set up, we can now also set a second theme on that material app besides this theme.

This is our light mode theme, but we can also now add a dark theme parameter to material app.

And this dark theme parameter can now be used to configure, well, a theme for dark mode.

For this, we can again use theme data.

And now conveniently, there actually is a dark constructor function, which we can use to get a lot of default dark mode settings and configurations so that we don't have to adjust everything manually to get dark mode to work.

So we can use this dark theme data constructor function instead.

And then here we can again use copy width and apply material three by setting this to true.

But then here of course, we can also set our color scheme and set this to the dark color scheme we just created.

And with that we're creating a dark theme here, which can be used as an alternative to the main theme.

Of course, that just brings up the question

which theme will be used.

And this can be controlled

with yet another parameter you can set on the material app.

And that would be the theme mode parameter.

Theme mode wants a theme mode object.

And here you can actually choose from

multiple predefined theme mode values.

You can, for example, force the app

to always run in dark mode

or always run in light mode,

ignoring what the user might have set up

in their system settings.

But you can also use theme mode system

to take a look at the system settings chosen by a user

and apply the appropriate theme.

And this is actually the default,

so you don't need to set this yourself,

but it is worth knowing about this setting.

Well, and with that, if you force a reload

and you put your phone into dark mode,

you should see that dark mode.

But here I'm now actually getting an error,

and I'm getting an error now

because in one of my widgets,

in the expenses list widget to be precise,

I'm actually referring to the margin setting

on my card theme.

And I'm telling dart that this will not be null

when actually now in dark mode it is null.

It is null because in dark mode

I am not setting that card theme.

That is why you should also be careful

with using this exclamation mark operator.

You instead might want to check whether this is null or not and then set different horizontal values based on the check result.

Here I'll solve the issue by simply copying my card theme from the light theme to the dark theme here, so that I am setting these margins again.

And with that, if I force a reload, the error goes away.

And you can see I have this dark background color on my scaffold now.

You will also see that the other styles don't look that great though.

And part of the reason for that is that the color scheme we're generating here uses a specific `seedColor`, but the shades and variants that are generated by `fromSeed` are actually optimized for light mode.

We have to tell our color scheme that the different shades and variants that are generated should be optimized for dark mode.

And this can be done by setting the brightness parameter in this `fromSeed` constructive function and setting this to `brightness dark`.

And this tells Flutter that this is a color scheme for a dark brightness, so for dark mode in the end.

If you now force a reload, this already looks much better because we have this dark app bar, for example.

But that being said, the card background color doesn't look that great though.

And the reason for that simply is that in the dark theme, I'm currently still using the normal color scheme

which I created first
for getting the secondary container color
off the card theme, and that's the wrong color scheme.
Here we wanna use the newly created dark color scheme,
which takes our dark colors into account.
If we save it with that,
these expenses already look much better.
Now of course, the buttons here in this modal
maybe also don't look that great.
And to change those we can again copy
the elevated button theme settings from the light theme,
the default theme, and add them to the dark theme.
But here, of course, we now also wanna use
the dark color scheme, which we created
and see how that looks like
if we take primary container from there
for our dark color buttons.
With that, this already looks a bit nicer.
Maybe the foreground color should be changed though.
And therefore here for the dark theme
for the elevated button theme,
we could also add foreground color
and set this to `kDarkColorScheme`.
And then `onPrimaryContainer`.
Because the primary container is the background color
for the foreground color
which colors the content inside of that button,
`onPrimaryContainer` might make a lot of sense
if you consider the name of it,
primary container and `onPrimaryContainer`.
So with that, if we save this,
yeah, that definitely looks better.

We could also change the color of this text button,
but I'm happy with that.
And with that, we now also added a dark theme.
Indeed we added it such that the device settings
are taken into account.
If a user switches back from dark to light mode,
our app adjusts accordingly.

32. Using Another Kind of Loop (for-in)

So at this point we're almost done
with this app,
and this was another long course module,
but again, with many important new concepts,
and of course a lot of chances
to repeat concepts we learned about before.
Now, the missing piece for this app here
is of course the chart.
And to build this chart here,
I first of all wanna go to my expense model
and add a new class, a new data model to this file.
And that will be the ExpenseBucket class.
Because for building this chart, we need multiple buckets,
one for every category,
so that we can sum up all the expenses
that belong to one category and display them in the chart.
So for that, I'm creating a separate class
that will hold that summed up data.
And this class here, which defines a group of expenses
in the end needs two main properties.
It needs a Category property, which is of type category.
So which uses our own Category enum type here.
And in addition to that,

I also wanna have a list of expenses,
because such a bucket, as mentioned,
will include multiple expenses.
That's the idea behind this class.
Therefore, of course,
we then can add our constructor, ExpenseBucket,
where we set this.category and this.expenses,
and both should also be required, like this.
Now, in addition to setting up those properties
to store all these expenses,
I also wanna add a getter to this class here,
which will return a double value,
so a number with decimal points.
And I'll name this totalExpenses.
And this is simply a utility getter you could say,
which I'm adding, which sums up all the expenses
we have in here,
so that we can conveniently get the totalExpenses
from a bucket.
And keep in mind we'll have one bucket per category.
Now, to build that sum,
I'll create a helper variable sum here,
which starts at zero.
And I wanna return this sum in the end.
But in between these steps, I wanna sum up all the expenses.
And this can be done with a for loop,
though I'll use a new version of that for loop,
a variant of that for loop, which we haven't seen before.
Instead of creating a for loop
where we create a helper variable i
and we then keep on looping as long as i
is smaller than expenses.length, for example,

and we increment `i` by one after every iteration.

Instead of doing that, which we could also do

to get hold of the individual expenses,

I'll use a variation of that for loop,

which is perfect for going through

all the elements in a list.

You can use this syntax where you create a helper variable,

which for every loop iteration is final

because it will be recreated after every iteration.

And here I'll create a helper variable `expense`.

And then we can use the special `in` keyword

to simply tell Dart that we wanna go through

all the items in this list, so `in expenses`.

And in every iteration, a new item will be picked

and will be stored in a newly created final variable

called `expense`.

And then therefore inside of that loop body,

so between these curly braces,

we have access to that single `expense`.

And this simply is an alternative

which is perfect for going through the items in a list,

since it is a bit shorter and maybe easier to understand

than manually managing a helper variable `i`

and using that as an index,

which you could of course also do though.

But here we can loop through all the expenses

in our `expenses` list like this

and then simply add the `expense` amount to the `sum`

by setting `sum = sum + expense.amount`.

And this can actually be shortened

and can instead be written like this,

which is the equivalent to setting it like this.

This is simply shorter
and tells Dart that you wanna set sum equal to sum
plus the expense.amount.
And this is therefore how we can define such a getter
that sums up all the expenses in this expense list
and returns a total sum as a number.
And we'll need this helper for the chart,
because in the chart, of course, the different bars
for the different ExpenseBuckets,
so for the different categories,
should be displayed in relation to each other.
And for that, of course, we need to know
the total expense of every bucket
so that we can adjust the chart bar heights.

33. Adding Alternative Constructor Functions & Filtering Lists

Now adding this getter
to ExpenseBucket is one important step I want to take
before we start creating the actual chart.
The other thing I wanna do here,
I want to add my own alternative named constructor function,
which can be done by repeating
the name of the class, ExpenseBucket,
and then adding a dot right after it
inside here of the class definition
and then adding the name of the extra constructor function
you want to add to this class.
Like in my case here, forCategory.
That's the extra name I want to use.
And this is how you can add additional constructor functions
to your classes.
And this is of course a feature we used

from time to time in this course.

For example, here on `EdgeInsets.symmetric`,
where we create such a `EdgeInsets` object
by using the alternative `symmetric` constructor function.

This here is how you can define
your own alternative constructive functions,
alternatives to the default constructive function.

So what's the idea behind this alternative then?

Well, my idea here
is that I do get a list of expenses here,
which could be named `allExpenses` here.

So this is simply a parameter accepted
by this constructive function.

So we expect to get a list
of all expenses here from all categories.

And I also want get a second argument
which sets the category,
which is one of the properties I do wanna set here.

And now the idea behind this extra constructor function
is that we add some logic to this constructor function
to go through all the expenses we got,
and then filter out the ones that belong to this category
so that we can set the expenses for this `ExpenseBucket`,
which is tied to one specific category
to a list of expenses that only are for that category.

So this is simply a utility constructor function
that takes care of filtering out the expenses
that belong to a specific category.

Now two said this `expenses` property to the `expenses`
that have this category at the point of time
this constructor function is called,
we have to add a so-called initializer list

by adding a colon
after our constructor function parameters here,
so after this closing parenthesis,
and then here you can set expenses.
So this property here equal to allExpenses,
so this list of expenses we're getting here,
and then filtered for the expenses with this category.
And this can be done with help of the built in where method.
Now this is a method provided by Dart,
which is available on all lists
and where is simply a method
that allows you to filter a list.
It takes a function as an argument,
so you pass a function here as an argument to where.
Here, an anonymous function using this arrow notation.
And then you automatically get your
individual list items as input.
So here I get a single expense since this function
is executed on all the expense items in allExpenses.
So every time it executes, I get one single expense.
And here we have to return true if we want to keep
that expense in the filtered list
or false if we want to get rid of it.
And here, of course, I don't wanna hard code true or false.
Instead, I wanna derive true or false dynamically
by checking whether the expense
I'm currently evaluating here has a category that's equal
to this category here.
So to the category property of that class.
So if the expense I'm currently taking a look at
has the category I wanna set for this ExpenseBucket,
I want to keep that expense in that list of expenses

that is stored in this bucket,
otherwise it should be dropped.
That's what we're saying here.
And now we just have to make sure
that we convert this back to a list.
And here we have to check with a double equal sign
because a single equal sign tries to assign a value,
which is not what we're doing here.
With a double equal sign, we check for equality,
which is what we want to do here.
And with that,
we got this utility constructor function in place,
which we will soon use to create different buckets
that are represented by chart bars in our chart.

34. Adding Chart Widgets

Now, with this newly added
ExpenseBucket class here,
we're now ready to work on the actual chart.
And for that, in the Widgets folder, in the Lib folder,
I'll add a chart sub folder,
and I prepared two new widget files,
which you'll find attached to this lecture `chart_dart`
and `chart_bar.dart`.
And you should download both
and place them in your Chart folder in the Widgets folder.
Now, these files are responsible
for creating a chart widget,
which then contains multiple `ChartBars`.
And I'll walk you through these widgets in just a second.
You should make sure that in `chart.dart`
these imports work for you

so that you're using your project name here
and that you have the same folder names.
I'm using widgets, models, chart, and so on.
Of course, you also must use the same file names.
And you should make sure
that if you named your model differently,
the ExpenseBucket or the expense model,
you update those names here in chart.dart.
After all, errors should be highlighted
if you have any in this file.
If you followed along with the same conventions I used
throughout this section though,
this file should work fine for you.
And the same should be true for chart_bar.dart.
Attached, you'll also find a link
to my complete current project snapshot,
so that you can also compare the other code files to yours
in case you're facing any errors
with those newly added files.
But what do these files do now?
Well, first of all, let's use them.
In expenses.dart, we wanna replace
this chart placeholder here
with the new chart widget that we just added.
So for this, we have to import chart.dart
from the Widgets folder,
and they are from the newly added Chart folder.
So that import here must be added.
And with that import added,
we can use the chart, which is here in this column,
and we must set the named expenses parameter
to our _registeredExpenses here.

So to all our expenses.

Because in the `chart.dart` file, we're then actually getting those expenses and storing these expenses in a property.

But I got a getter, a helper getter,

which then creates these `ExpenseBuckets`.

Using this newly added `ExpenseBucket` class

on which we worked over the last lectures

to create different buckets for different categories,

so that we have these four buckets that are needed

to display these four bars or up to four bars in the chart.

Because if we save everything now and force a reload,

we should see this chart here.

And if we switch to dark mode,

which we, of course, also can do,

this chart should also look good in dark mode.

Yeah, so this also looks quite nice.

But how is this being displayed?

Well, we got these buckets here, as mentioned,

which are generated in this getter,

and this getter is in the end used down there

in this widget tree to go through all these buckets

and create these `ChartBars`.

Now, this code here might look strange,

though I have mentioned it before in this course already,

but it's still worth repeating what this code does.

As I annotated here, it's an alternative

to the built-in `map` method.

And you can use this code here in your widget tree

if you're inside of a list of widgets.

So if you're in a place where a list of widgets is required,

like here for the children of a row.

Then you can use this for loop here

with a syntax that's similar to what I showed you
in this ExpenseBucket class in this getter here
where we also had such a for in loop.

But if you're using it in a widget
or in a place where a list of widgets is expected,
you actually have no curly braces thereafter,
instead you just have this for in loop like this
to loop through a bunch of pieces of data,
and then right below this for loop syntax here,
you have the widget that should be created
for every item in that list of data.

So here I'm creating one ChartBar widget
for every bucket in my list of buckets.

And buckets, keep in mind, is this getter here
where I return this list of four ExpenseBucket instances.

So I'm looping through these buckets here
and I'm creating one ChartBar in this list
that belongs to this row for every bucket.

Now, the ChartBar has one fill parameter,
which is set to a number.

And the number is based on the total expenses in a bucket.

So that is this totalExpenses getter being used here.

So the sum of all the expenses that belong
to a certain category, divided by maxTotalExpenses.

And maxTotalExpenses is actually a getter
that belongs to this chart widget.

In this getter, I go through all my buckets
with another for in loop,
and then I simply determine the highest total expense
from all these buckets, so from all these four buckets,
in this case here.

I do that here by taking a look

at each bucket's total expenses,
so the sum of all the expenses in that bucket,
and if that's higher than the highest total expense
I got up to this point starting at 0,
I take that bucket's total expenses
as the highest amount of total expenses.

And I'm doing this because in the end, in the chart,
the goal, of course, is to put these different bars
in relation to each other.

And the bucket with the highest expense
should have the tallest ChartBar
and the other buckets' ChartBars should be drawn
in relation to that.

So that's what I'm doing here with this getter.
And I'm simply passing the division of the total expenses
of a single bucket by the maximum total expense
across all buckets to ChartBar,
because in chart_bar, we're using this fill parameter
and this fill property here
to set our ChartBar height.

And I'm doing this with a new widget,
the fractionally sized box,
which simply does not take a fixed height in pixels,
but a height factor instead,
which simply is a number between 0 and 1.
And 1 means that it takes 100% of the available height,
0 means it takes 0%.

And that, of course, is a very helpful widget
for setting different ChartBar heights therefore.

Besides that, I'm not doing anything fancy
just setting some colors based on my theme.

Though here, it's worth pointing out

that I'm actually dynamically deriving
whether we are in dark mode or not
with help of this line of code,
to which I'll get back in just a second.

But other than that, I just got some default styling here.
And in `chart.dart`, I'm also not doing anything fancy
besides setting these `ChartBars`.

I just got more widgets to also display these icons
below the `ChartBars`, that's what I'm doing with this row,
where I'm mapping all my buckets to icons, essentially,
using my `categoryIcons` map,
which we created way earlier in this course section.

This map here in `expense.dart` in the `Models` folder,
I'm using that here to set the Icons below the `ChartBars`.

And I'm also dynamically finding out
whether I'm in dark mode or not here as well.

So what's up with this line of code then?

In flutter, you can use the built-in `MediaQuery` class
combined with `of` context to get some information
about the environment in which your app is running,
so to say, and we'll see `MediaQuery` again
in the next course section.

Here we're using it to find out what the current brightness
of the platform is, if it's dark or light,
so if we're running in dark mode or in light mode.

And I'm doing this because down here,

I actually wanna set different colors
based on whether I am in dark or light mode.

Of course, we can set up totally different colors
in our themes, but here I'm actually using different colors
of the generated theme color schemes as you can see.
Secondary in dark mode versus primary in light mode.

And that's therefore, a never useful pattern to know.

You're not limited to just setting some colors
in your themes and then using those colors everywhere.

Instead you can set up those colors,
but if you then have a case like we have it here
where in dark mode you want to use the secondary shade
that was generated, whereas in light mode,
the primary color looks better,
you can dynamically find out in which mode you are running
with help of MediaQuery.

And I'm doing this here in the `chart.dart` file,
but also in the `chart_bar.dart` file.

Here I'm also dynamically deriving
whether we're running in dark mode or not
to dynamically use a different color
for the color of my decorated box here,
which in the end defines the color of the `ChartBars`.

And that's therefore, a never theme related pattern
that's worth knowing that you can use different settings
based on whether you're using dark or light mode
directly in the widgetry as we're doing it here
with help of MediaQuery.

And that's also how such a chart could be rendered.

You also saw this for syntax here
for outputting a list of widgets,
which is similar to this for syntax we used here
in a non-widget place.

It's almost the same,
but we don't have the curly braces here.

And as mentioned here, it's simply an alternative to `map`
when using it here in a place
where a list of widgets is wanted.

You see the alternative down there actually.

Here I am using buckets map,
and we could indeed replace this with for in here as well,
just as we're doing it here.

But that's therefore, how we can add such a chart,
which works in dark and in light mode,
and how we can in general build such an expense tracker.

35. Module Summary

And that's now it for this course section.

Another long course section,
but a course section with many new crucial concepts.

We did, for example,
learn how we can show things like a snack bar
or a modal bottom sheet,
or how we can handle user input with text fields
and with dropdown buttons as we're doing it here.

How we can, for example, build such a manual date picker
by showing a date picker with the built-in
show date picker function that is provided by Flutter.

We learned how we can show a dialogue,
and along the way we learned
that this context object is important
for all these methods since it carries some meta information
about a widget

and the widget's relation to our widgets that are required
by Flutter to show all these extra elements.

You learned how you can handle user input
with text editing controllers
or by manually storing the input values
as we're doing it here for a selected date
and selected category.

You learned that if you're using text editing controller,
you also must dispose of it with help
of the built-in dispose method into which you can hook
to make sure that the memory
for these controllers is freed up as they are removed.
You, of course, also learned about many other new widgets
in this course section here,
like, for example, the icon button or app bar,
which we used in this app.
You also practiced many of the patterns
and things you learned in previous course sections,
like passing pointers at functions to your own widgets
so that some child widget can call a function
that belongs to a parent widget.
You learned about useful widgets like the list view widget
with its builder method,
which is great for building lists
of unknown length in a high-performance way.
You learned about the dismissible widget,
which is great for easily making list items dismissible,
as the name kind of suggests, I guess.
And you, of course, also learned about theming,
how you can define
and use different color themes,
how you can define different themes for light and dark mode,
and how you can make sure that you're using built-in themes
and you just copy them
and overwrite exactly the settings you wanna overwrite
so that you don't have to set everything up from scratch,
though you can also do that if you want to.
And with that, you now, of course,
got even more important tools in your Flutter tool belt

that will help you build exactly the apps you wanna build.

But of course, we're also not done yet.

We can and will dive in way deeper.

5_expense-tracker_responsive-addaptive-ui

1. Module Introduction

In the previous course section,
we built this expense tracker app here
where we can add new expenses, delete expenses,
and see our expenses in a chart here, which updates
whenever we change our expenses.

In this course section here, we're going to build up
on that project and make this application
that we built there, responsive and adaptive.

And for that, in this course section here,
you are of course going to learn what exactly responsive
and adaptive means.

You will learn how you can change the layout
of your app based on the available screen sizes
that you have available on the devices
where your app runs.

You will learn how you can detect and use screen
and platform information
and how you can build adaptive widgets,
and what adaptive widgets are in the first place.

2. What is "Responsiveness"?

Now to dive into the concepts of responsive
and adaptive apps and widgets,
I'm back here in this expense tracker project
on which we worked in the previous course section.

And attached you find my lib folder,
which you could drag and drop into a new project
in case you did not follow along in the previous section.

You should then also make sure that you install the uuid and intl packages into that project, though, so that these dependencies are added to your pubspec.yml file.

And with that, you are ready to follow along.

So what's this course section all about?

What are responsive apps and adaptive widgets?

Well, with responsive app,

I mean that if I have a device

where screen rotation is unlocked, that's important,

and I then rotate my screen,

the app does generally adjust,

but, of course, the UI is not optimized for that.

Here you can see that this UI now looks awful.

I have this chart, which is way too big,

and I have my list of expenses here,

which, yeah, is scrollable,

but which takes up only a small chunk

of the available space.

On the other hand, if I try to add a new expense,

this screen here is all right,

but we also could organize this screen differently

because with all that available width

that we have here in landscape mode,

we could, for example,

position the amount next to the title

and the dropdown next to the date picker

or something like this.

So the UI is simply not optimized

for this screen rotation.

That's the problem.

And a responsive app is an app where the layout

and styling adjusts to the available space,
to the available width,
and to the mode in which the app is run,
so to portrait or landscape mode.
That's what responsiveness means
and what we're going to take a closer look at
in this course section.

3. Locking the Device Orientation

So let's see what we can do
to optimize our app for different screen orientations
and available widths and heights.
And one thing we could of course do
is that we could disable this landscape mode altogether.
We could simply ensure that our app never adjusts
when the device is rotated,
so that our app can effectively only be used in one mode,
in portrait mode, for example.
That is something we could do
and it is something we can do in flutter as well.
And to do that in flutter,
we first of all have to import something
in our main.dart file,
so where this main function is.
And here we have to import a package, the flutter package,
but from the flutter package,
we have to import from the services dart file.
Which is a file we haven't used up to this point,
but this is a file that exposes some functionality
that we can, for example,
use to lock the screen orientation of our app.
And we can use a feature from the services dart file

in our main function

before we even run the app.

Here, we can use the system Chrome class

provided by flutter in the end,

and there set the preferred orientations

by calling set preferred orientations.

Now set preferred orientations

then takes a list of all the device orientations

we want to allow.

And setting such orientations

will then effectively prevent our app

from adjusting to other orientations

that are not specified in this list.

So here in this list,

we can then access the device orientation object here

or the device orientation enum to be precise

and access the different orientations.

And for example, here we can set the preferred orientations

to portrait up and only portrait up.

Now with this code added here,

you then have to chain a method after it, the then method,

because set preferred orientations

actually gives you a future,

and I covered futures in the previous course section.

In the end, it's a object that provides a then method,

which then wants a function as a value

which will be executed once the future

is done with a certain task.

In this case,

once the device orientations have been locked in, so to say.

Now in here in this then method,

you should therefore provide a function,

in this case an anonymous function as input.

And this function actually receives another function as input itself.

But we don't need to worry about that since we don't need to do anything with that function.

We just have to accept it here.

And you should then move your run app code.

So this entire run app function here into this then function.

So into this function you pass to then, so that the app basically only gets started or the UI only gets applied once the orientation has been locked in.

With that code added,

you should also add another line of code before you call set preferred orientations.

And here you should reach out to widgets, flutter, binding, and call ensure initialized on that.

This is required to make sure that locking the orientation and then running the app works as intended.

If you now force a restart, or better yet you stop your currently running session, and run the app again without debugging here, you should see that now once the app started, if you rotate the device, it does not rotate anymore.

I mean the device of course rotates this emulator, but the app in there is stable and does not change.

So this now works as intended here and we force the user to use the app

in exactly the way we want them to use the app.

Now locking the device orientation is one way of ensuring that the app is used as intended.

And for some apps,
this is the best way of ensuring that the app
and the UI works,
because some user interfaces simply can't be adjusted
to work in landscape mode.
But for this app we can indeed adjust it accordingly.
And therefore for this app,
I'll actually get rid of the code I wrote before.
I will comment it out here.
I will not log the device orientation
and I'll also get rid of that code down there,
therefore, of these closing parentheses,
and get rid of this import.
I will get rid of all of that and stop this running session
and restart the app
to make sure that it really works in all modes again now.
Because the next step now
is to make sure that our UI does adjust to landscape mode
and different device orientations.

4. Updating the UI based on the Available Space

Now to adjust our app UI
to different orientations,
let's first of all take another look
at this landscape mode UI.
What's wrong with that here?
Well, the problem is that this chart is too big
and this list doesn't have enough space, therefore.
A better UI here would be
to have the chart on the left
and the list to the right, for example.
And of course, that's just one possible way

of improving the UI.

Definitely feel free to use this as a practice opportunity to build different UI variations for different orientations.

But here I want to indeed move the chart to the left and the list to the right.

The question now just is, how do we do that without breaking portrait mode?

Because of course here in portrait mode, I want to keep the UI like this.

I only wanna adjust it if we're in landscape mode, or if we have that much width available to be precise.

On an iPad, for example, we also might want to have the layout on the left and the list on the right, even if it would be in portrait mode.

So it's really the width I care about.

If I have a lot of width available, having chart next to list might be preferable to having the chart above the list.

So how can we adjust the code dynamically to the available space and update the UI if that space changes, for example, because the device was rotated?

Well for that we can go to expenses Dart because that is in the end where

I do output the chart and my list here, this main content in the end is the list if we have some list items.

And therefore it's in the end here where we wanna switch from a column to a row if we are in landscape mode, because that would ensure that items would be rendered

next to each other instead of above each other.

With a column they are rendered above each other,
with a row, they would be rendered next to each other.

So therefore what we need to do here
in this build method of expenses Dart
is we need to find out how much width we have available.

And if it's enough width,
we wanna switch from a column to a row.

Now to find out how much width we have available,
we can use the media query class provided by Flutter.

And you briefly saw media query
towards the end of the previous course section.

Media query has an of construction function,
which wants the context object as input
since that holds important information,
meta information to be precise,
about the widget in which this code executes
and its position in the overall widget tree.

And once media query has been connected
to context like this,

you can access various pieces of information
on that created media query object.

For example, there is a size property which you can access,
and this size property gives you access to a size object
which holds more information
about the available size on the device.

For example, you can get the aspect ratio
of the screen on which your app is running,
but you can also get the height and width,
and that simply is the available width and height
on the device in which your app is running.

So here if I access width, for example,

I get the width of my screen.
And if I simply print this here
so that we can take a look at that
and we maybe also print the height here,
we can see that if I save this and I force a reload,
and we then open this debug console,
that we see the width and height being output here.
And of course the exact numbers will depend on the device
you are simulating this app on,
but this here would be the width and height
of the virtual device on which this app is running.
Now if I rotate this, you will see
that it automatically prints new values here,
which are the same values as before, just swapped,
because now the old height is the width
and the old width is the height
because we switched from portrait to landscape mode.
And since this gets printed again here,
we also see that the build method seems to execute again
if you rotate the device.
And indeed this is something Flutter does for you,
it executes this build method again
so that with the media query,
you get the updated width and height
so that you could adjust your UI
that's returned by that build method,
so the widgetry that's returned by that build method,
to the available width and height.
Therefore now that we know that,
we can store that width in a variable
just to make working with it a bit easier.
And now we could use that width

to render a row instead of a column
if we have enough width available.
For this, we can use any
of the conditional rendering approaches we learned about.
For example, we can use a ternary expression.
And we can use this with variable,
which I just created, this one here,
down here in the body of this scaffold widget.
And maybe check if width is smaller than 600, let's say.
And if that's the case, we wanna render a column
so that we have the chart above the list
because we don't have a lot of width available,
hence having this vertical layout makes more sense.
On the other hand,
if the width is equal to 600 or greater than 600,
we want to switch to a row instead.
So we then we want to render a row here.
And of course the children of the row
are the same as before for the column.
So we can copy those children
into the children list of the row.
And with that we switch from a column to the row
depending on how much width we have available.
Now there's just one thing we have to do
because right now this actually fails
and we see nothing on the screen
because one thing we have to adjust now
is related to our chart.
This chart widget in the end renders a container
that has a width set to double infinity.
Now I'll get back to what exactly the problem with that is,
but in a nutshell, the problem is that

with this width setting here,
this container simply tries to take up
as much width as it can get,
only restricted by the parent widget
in which this container,
so in which this chart in the end, is included.

The problem with the chart being inside of a row now is
that the row also has a width of infinity,
so it tries to get as much width as possible.

And if you have a child
that tries to get as much width as it can get
and you have a parent that tries to get
as much width as it can get,
you get an error because Flutter
is not able to display a UI.

To work around that,
you should wrap the chart also in expanded,
which is basically always the solution
whenever you have widgets nested into each other
that don't work together like this.

And it should work.

So if you save this and force a reload,
you now see that indeed the chart sits next to the list.

And with that, we now got a UI that makes way more sense
because now this UI works here on landscape,
but it also works on portrait mode.

And you see that as I switch between those modes,
the UI gets updated
and gets adjusted to the available width.

And that's exactly what response of the sign means,
your layouts adjust to the available sizes.

5. Understanding Size Constraints

Now before we continue
with adjusting our app to different device sizes
and we, for example, work on this modal
and how it looks like in landscape mode,
there's one thing I want to take a closer look at.
And that would be widget size constraints,
or specifically the issue we had before
with the chart not working inside of a Row.
I want to make sure that you really understand
what exactly the problem was with that
and why we need Expanded here.
And for that it's really important
to understand the concept of size constraints in Flutter,
and to understand how in general Flutter widgets
get sized and displayed on the screen.
Because it's important to understand
that widgets actually get sized
based on their size preferences
and the size constraints applied by the parent widget.
And to make this sentence a little bit less abstract,
let's take a look at the Column widget.
This Column widget has some preferences
regarding the size it wants to occupy.
It wants to take as much height as it can get,
but it only needs as much width
as defined by its largest child,
So as defined by its children overall.
But since the Column, in theory,
of course, wants to take as much height as possible,
it's important that the Column widget is inside of a parent
that restricts the height because otherwise

it would go outside of the screen boundaries,
which is not the goal.

That's why we, for example,
have no problem if a Column is used directly
inside of a Scaffold widget.

So as a value for the body parameter of a Scaffold widget
because the Scaffold widget turns out
to constrain the height and the width
to the maximum device height or width.
But every widget has different constraints
for its child widgets.

So this is just an example here.

But for this example,
since we have the Column inside of a Scaffold
with those constraints,
the result would be that the maximum Column height
is actually the maximum device height
because that is the constraint by the parent widget,
and the constraint of the parent widget
overwrites the preferences of the child widget.

That's how sizing generally works in Flutter.

And you can actually see this
if you open the Flutter dev tools.

Here, I'm doing this in a browser.

Now if we open this Flutter inspector,
we can browse this widget tree
and select different widgets there.

And if we, for example, select this Column,
which is right below the Scaffold,
we can see that here on the right
we get this layout explorer.

And this layout explorer, for example,

tells us the height and the width
that was assigned to this Column widget,
but it also tells us
the constraints this Column widget get from its parent.
So in this case, from the Scaffold.
And here, for example,
we see that for the width the constraint was that anything
between zero and the maximum device width is allowed.
And for the height it's the same, zero and maximum height.
And therefore the Column widget
actually chose the maximum height and the maximum width,
though the width simply depends
on the child widgets inside of the column.
Because as mentioned before, for the Column widget,
and actually for most Flutter widgets,
the actual height or width that is taken
depends on what the child widgets want.
Though for the Column widget, there is something special
to be noted regarding the height constraints
it passes down to its children.
So to the Chart, for example,
which is inside of the Column.
We can see those constraints in here.
Here we see that it's unconstrained vertically
so that the Column widget does not pass
any height constraints to its child widgets,
so to the widgets that are passed to the Column.
And that, for example, would be a problem
for the ExpensesList
because this in the end renders a ListView,
which is also just a list and kind of a Column therefore.
And indeed the ListView

by default also would have no constraints.

We don't see it here when we select the ListView,
but actually if we select the child inside of the ListView,
this Dismissible widget,
we can see that for the Dismissible widget
its height is unconstrained.

And since it is inside of the ListView,
that of course means that it's the ListView
that's not constraining the height for Dismissible.

So ListView has a unconstrained height as does Column,
and therefore ListView would not be restricted
in the height it tries to take up.

And therefore it would of course go out of the boundaries
of the device screen on which the app is running
because there simply is no height constraint.

And that's exactly why you need this Expanded wrapper here
because Expanded does something special.

If you add Expanded as we're doing it here
around the ListView,

in the end, Expanded does set height constraints.

It sets the maximum height of its child widgets,
so of the ListView in case of this Expanded widget here,
to the maximum available height.

Instead of getting as much height as possible,
so a height of infinity in the end,
it takes as much height as it can get.

And that's an important difference
compared to an infinite amount of height,
which is what Column and ListView would set as constraints.

Expanded narrows that down
and limits the available height to its child widget,
so to the ListView in case of this Expanded widget here

to the actual available height,
that's why wrapping unconstrained widgets with Expanded
makes sure that unconstrained widgets
inside of other unconstrained widgets,
as we have it here with ListView inside of Column,
doesn't cause any problems anymore.
That's also the problem we fixed here with the Row and Chart
because Chart actually does not render a Row,
but a Container where I explicitly set
the width to double.infinity
to make sure that this container takes
as much space as it can possibly get.
But of course that's a problem
if we do this inside of a Row,
which also has a width of infinity.
With Expanded, as I just described,
this is fixed because expanded between Row and Chart
makes sure that chart is actually limited
and can only take as much width as is actually available.
And that's how widgets are sized
and what size constraints are all about.
This in the end, explains why certain widgets
can be used inside of other widgets without problems
and why, for example, a Column inside of a Column
could cause a problem,
but those problems then can be fixed, for example,
by adding an Expanded widget between the two Columns.

6. Handling to Screen Overlays like the Soft Keyboard

So now with this exploration of sizing
and constraints out of the way, I wanna get back
to making this app a bit more responsive.

And for that, we of course already made sure
that we have the chart next to the list,
but now I wanna work on this modal.

Now this actually does also work in landscape mode,
but I would say we could use this space a bit better
if we are in landscape mode because we, for example,
don't need this long title row here.

In addition, of course, we have the problem that right now
if I open my keyboard,
I actually can't reach the other inputs,
and that therefore is also something we might want to fix.

It's also the reason why I'm getting an error here
in Visual Studio Code after opening the keyboard here
in this landscape view of this modal.

Now, first of all, let's make sure
that we can open this keyboard
and actually reach all inputs.

To achieve this, we have to go to `new_expense.dart`
because that is in the end where we do render this form.

And in the build method of our new expense state class here,
we can again use `MediaQuery.of context`
as we did it before,
but now not to calculate the available width or height,
at least not yet.

But instead on this `MediaQuery` object,
we can also access a `viewInsets` object.

And this is an object that contains extra information
about UI elements
that might be overlapping certain parts of the UI.

For example, if I access `viewInsets.bottom` here,
I get any extra UI elements
that are overlapping the UI from the bottom.

And that's of course exactly what this keyboard does.

It slides in from the bottom and over overlaps my UI.

So here with `MediaQuery.of(viewInsets.bottom)`,

I can get the amount of space taken up by that keyboard.

I can then save this in `keyboardSpace`

or whatever you want to name it

and now we could use that to adjust our padding here

and actually add more padding from the bottom if this space

is taken up by the keyboard.

So we could set the bottom padding here

to `keyboardSpace + 16` so that we always have

at least 16 pixels of bottom padding,

but we add that extra space

that might be taken up by the keyboard.

Now if we do that, we have to remove `const` though,

since `EdgeInsets` is now calculated dynamically.

Now, if you do set the padding like this

and you save everything,

you'll note that nothing changes though,

we still get this error.

The reason for that is that we also must make the content

of this modal sheet, so this widget tree that is returned

by the new expense widget scrollable,

so that we actually can scroll away

because if you take a look at the screen,

of course this must be scrollable,

otherwise we can't reach the other input fields.

Therefore, we should wrap this padding widget

with another widget.

So I'll choose that refactor action

and choose the `SingleChildScrollView` here,

which you also saw before in the course already.

Now if you do that and you save everything and reload,
you'll see that this looks much better.

Now this is scrollable and we can reach all the inputs.

Of course, it's still not ideal,

but on landscape mode, we'll have to live with that

because we do have that keyboard there

and there is only a limited amount

of height left thereafter, it is what it is.

But now at least this looks better.

Though you will note that if you close the keyboard,

the modal sheet now actually does not take up

the entire height anymore, and you might be fine with that,

but you can also make sure

that it does take up the entire height again.

By the way, here, I'm still getting an error

that there is some overflow,

but this seems to be some weird behavior or bug

since I can't detect any overflow here.

I can reach everything and everything seems to work here,

so it looks like this is kind of a buggy error message here.

Either way, to make sure that we take up

the entire height again,

I'll wrap this `SingleChildScrollView`

with another widget.

I'll wrap it with a `SizedBox` to be precise

because on that `SizedBox`, I can set the height

and I'll set it to double infinity

to make sure that this takes up as much space as it can get.

And with that, if we save this and reload,

I get this full screen mode again.

Still scrollable, but full screen.

And this strange error message,

which I'm going to ignore here.

But that is how we could make sure
that we can always reach all the contents
no matter if we view this application here in landscape mode
or as before in portrait mode.

There, of course, it also still works as it should.

7. Understanding "Safe Areas"

So now that we're making sure
that we respect the keyboard, and we make this scrollable
and add some padding at the bottom
if the keyboard is opened,
we also might wanna take another look
at these device elements like the camera,
which could be overlapping our content.

We fixed this in the previous section
by adding some top padding here,
and of course this works, but the problem is
that we have to pick an amount of padding here
that will probably work on all devices.

And whilst we can, of course make some guesses here
that are likely to work,
this is not necessarily the ideal approach.

Instead, there is a better way of solving this.

And for that, I'll first of all set this back
to the actual amount of padding I'd like to have,
no matter if there is some camera in the device or not.

But with that, of course you'll notice
that now the camera does overlap the title,
but before we'll fix this,

I wanna take a look at another screen.

For example here this starting screen.

Here it looks like Flutter automatically makes sure that the title and this plus button here are not behind that camera or that system bar.

So Flutter seems to have some built-in mechanism that makes sure that we stay away from these device features.

Unfortunately, that just does not seem to work here for this modal, but we can make it work there.

When we open that modal from inside the expenses start file here in show modal bottom sheet, we can add an extra parameter which is called Use Safe Area.

And we can set this to true.

A safe area is a simple feature built into Flutter that makes sure that we stay away from device features like the camera that might be affecting our UI.

And widgets like the scaffold widget automatically use this safe area concept internally.

So in most cases you don't have to worry about it.

But for the modal overlay, at least at the point of time where I'm recording this, Flutter does not by default use this safe area concept, but we can unlock it by setting this parameter.

And if we do that, Flutter will take a look at the device on which the app is running, and it will know how much space is occupied by the camera that might be embedded in the screen, for example.

And therefore, now if I click this plus button, you see that the modal stays away from that system bar.

And that's of course an enhancement because it makes sure that the app looks better,

and we don't have to guess how much padding
we need to add to the top.

8. Using the **LayoutBuilder** Widget

So now with all these improvements
out of the way,
let's come back to making our app more responsive,
especially here in landscape mode.
This modal now works and looks better than it did before,
but I also mentioned before already
that we might want to move the amount input here
into the same row as the title input,
and then this dropdown button
into the same row as the date picker.
And we could do this inside of `new_expense.dart`,
by again using the media query,
just as we did it in `expenses.dart`.
There, we also took a look at the media query
to render a different UI based on the available width.
We could replicate this approach here in `new_expense`,
and it would be a fine way of solving this problem,
but I'll actually go for a different solution.
Because besides `MediaQuery`,
Flutter also gives you a specific widget you can use
for building a UI, a layout,
that automatically adjusts to the available space.
And for that here, in `new_expense.dart`, in the build method,
I will actually return a different widget,
instead of the `SizedBox`.
I'll return the `LayoutBuilder` widget,
which is built into Flutter.
And `Layout Builder` is a widget

that takes a builder parameter,
which actually wants a function as an argument,
a function which will automatically get the context object
and a constraints object.
So therefore here, I'll register an anonymous function
where I get this context and this constraints object.
And it's now inside of this function,
where we should return this SizedBox
and this entire widget tree.
So I'll cut it here,
and return it here inside of this function,
which we pass to the builder parameter of LayoutBuilder.
Here we go.
With that, if you save this and force a reload,
you will see that nothing changed.
If I open this modal, it looks and works as it did before.
But now inside of this function here,
which we passed to the builder parameter,
we get access to this constraints object.
And if we print this constraints object,
or to be precise, some of its properties,
we can see what's inside of that constraints object.
Because this constraints object is all about telling us
which constraints are applied by the parent widget
in which we use this widget here,
so the parent widget of the new expense widget.
And this constraints object tells us
the minimum amount of width we can use
and the maximum amount of width we can use,
and then the same for the height, min height and max height.
So if I save this and reload,
you will see that as I open this modal,

and I opened that debug console, I see the values here.

And here I actually only have three values,

because for min and max width it's the same value,

I have that fixed without any choice.

Now, how can these values be helpful?

Well, with them you know exactly

how much space you have available,

and you can then decide which layout should be rendered.

And this, for example, allows you to build a widget

that can be used anywhere in your widget tree,

and it does not care about the available width

or height of the screen,

it does not care about the screen orientation,

but it only cares how much width and height is available

to this specific widget here.

Now of course here, for this new expense widget,

since it is inside of that modal,

it is essentially the screen dimensions

that are available to us here.

It's the width and height of the screen, essentially.

So therefore we could also just use MediaQuery,

which does give us the screen width and height.

But we could now also use the new expense widget

anywhere else in our widget tree,

and it would only care about the constraints set

by the parent widget, it would not care

about the available screen width or height.

And we can then use this information here

to dynamically change our layout.

So for example here,

we could store the maximum amount of width

that we can use in a width variable,

and now use that variable,
just as we used variable here in expenses.dart
to render a different combination of widgets
based on the available width.
And LayoutBuilder will call this builder function again
whenever the constraints change.
So if the widget gets moved somewhere else,
or if we for example, change the device orientation.
So therefore, this will be updated
if the available min and max with values change.
And now we can use this width information
in this column here in this SizedBox
to render the content of the column,
the widgets in that column,
so the rows of that column, conditionally.
And we can do this by using a special syntax
offered by Dart, that can be used in lists like this,
where you write if, and then your condition,
for example, that the available width is greater
or equal than 600.
And then in the next line, you add the item to the list
that should be added if this condition is met.
So in this case, for example, a row,
so that we can have multiple widgets next to each other
in this column.
And you can also add a else statement
to define which widgets should otherwise be added
to this list.
So this is a list-specific if-else syntax
where you don't have curly braces,
but instead just if with your condition,
and then the widget that is added to the list

if the condition is met,
and else, the widget, if the condition is not met.
So with that, we are adding a row with multiple widgets
if we have enough space,
and otherwise we only have a single text field
in that first column row here.
Now, if I do have enough space
for multiple child widgets in the first column row,
I still wanna render my text field here,
so I'll copy and paste that.
And we could outsource it into a separate widget
to reuse some logic, instead of copy and pasting it here.
But for the moment, I'm fine with copy and pasting
to simply show you this general idea
of using `LayoutBuilder` to build dynamic layouts.
And therefore then, here in this row
after this title text field, we could add a `SizedBox`
that adds some width as spacing,
before we then add our amount widget,
which is this text field.
And this is wrapped in an expanded widget,
because text fields are actually widgets
that are also unconstrained regarding their width,
as is the row.
And as you learned before,
if you have an unconstrained widget
inside of another unconstrained widget,
you would face problems.
That's why we should actually keep this expanded widget here
since we're again in a row, and then add it here,
and actually also add `Expanded` here
for this first text field,

since this would otherwise also cause us problems.

So with that, we have this row with the title text field,
some spacing, and the amount text field,

if we have enough space,

and otherwise, if that's not the case,

we just have the title text field in the first row.

Therefore here, if I save that

and force a reload for example, you will see

that now Title and Amount are next to each other.

The styling is a bit off, but we'll fix this soon,

but they are in the same row.

And on the other hand, if I rotate this,

you see that they are still in different rows.

So that's how you can build different layouts

for different device sizes.

Of course, we still need to fix some things here,

because right now I have two amount fields here,

for example, and the styling also is a bit off.

Now regarding the styling, we can go to this row here

and add some cross-axis alignment here,

which in case of a row is the vertical alignment,

and set this to `CrossAxisAlignment.start`

to make sure that all the inputs in the first row get pushed

to the top, so that this now looks better.

And of course, we also wanna rearrange the second row here,

and get rid of that amount,

and instead move the dropdown button here

if we have enough width.

So therefore, we can go back to our widget tree here.

And after this first row here, with this if statement,

where we render the row or the title text field,

we can go to the next row

and render this entire row conditionally,
and check if width is greater or equal than 600.
And if it is, I want to render a row
that contains my LEISURE dropdown and the date picker,
and if it's not, I wanna render this row
which contains the amount controller and so on.
So therefore, this here is my else case,
but now we also need to define the layout
that should be rendered if there is enough width.
And for that here,
we then want to take this dropdown button.
So copy that,
and again, you could outsource it into a separate widget
and arguably that might be preferable,
but here I'll just copy it,
and then add a SizedBox maybe again for some spacing,
before we then add the date picker.
So here we can now grab this date picker,
which is wrapped by this expanded widget
since it is inside of a row,
and paste this here after this SizedBox.
So with that, I now also have my row
that should be rendered if we have enough space.
So if we save this and force a restart,
we indeed do have the dropdown next to the date picker here.
Of course, we also still have the dropdown here,
and therefore of course we'll have to fix this
and get rid of it, if we do have enough width available.
So therefore, if we go down here to this row
where the dropdown button is included next to the buttons,
we also want to add another if check
with the same condition as before.

If we have enough width, we wanna render something else,
else, we want to have this row here.

Otherwise, if we have enough space,
we wanna render a different row.

And in this different row here,
we don't wanna have the dropdown button,
instead there, we just want to have the other buttons.

So we could grab this spacer, and then these buttons,
and move those here into this list.

So with that, if we force a restart,
this is how this modal could look like
if we are in landscape mode.

If I go back to portrait mode, we got the old layout.

And that is how you can create a dynamic layout
with help of `LayoutBuilder`.

Though as mentioned before, in this scenario,
where this new expense widget effectively does take up
the entire screen anyways,
we could have also simply used `MediaQuery`
as we did in the expenses widget.

But I also wanted to introduce you to `LayoutBuilder`,
which can be very useful if you have a widget
that should only care about its parent widget,
and not necessarily about the entire available screen size.

Of course, all as mentioned before,
here we have a lot of code duplication and copying,
and this can make this code here quite hard to understand
and read, since we have all that duplication.

For that reason, you might want
to actually outsource these different widgets,
which are reused in different parts of the layout,
into separate custom widgets.

Here, I'll just keep it like this,
but it is something to be aware of
and to potentially improve.
But with that, that is how you can use LayoutBuilder
to also build a layout dynamically.

9. Building Adaptive Widgets

Now to come to an end for this section,
I also briefly wanna dive into
how you can build adaptive apps.
And for that of course it's important to understand
what this means.
In the end, it means that your UI
can also adapt to the platform
on which it's being presented.
And in general, you can use
exactly the same widgets and styling
no matter if you are running your app on Android or iOS.
But you can also adjust some widgets or styles
if you want to.
Now, to show you what I mean,
I am launching my app here on an iOS simulator,
which I started on my device.
Since I'm on macOS here, I can indeed build my Flutter app
for iOS on this device.
So that's what I'm currently doing here.
I started this app,
this same app on which we worked
throughout this and the last section on iOS.
And as you can see,
this app also looks totally fine on an iOS device.
And there are even some adjustments,

like for example, this tile here,
which is now centered,
which are made automatically by Flutter
as your app does run on a different device.
On Android, this title was on the left,
on iOS, it's automatically centered.
Though you could overwrite that if you wanted to
by going to your scaffold app bar
and setting center title to false, for example.
This would force the title to be on the left.
But it is of course also something you can keep
since these are nice touches
that make the app feel native
for the platform on which it is running.
You also might note that
the font is a different one
because by default here on iOS,
Flutter uses a different font family for the text.
But the general look and feel, of course, is the same.
And that's a good thing
because that is what allows you
to build your app once, to write your code once
and to use the same code and styling
for different target platforms.
But there also are some cases
where you might want to have a more platform specific look.
For example here, if I click save expense,
this dialogue, which I get here
is the exact same dialogue with the exact same look and feel
as I had it on Android.
And this might be fine,
it definitely does not look awful and it works.

But you might want to use the native iOS dialogue
look and feel here.

And to help you achieve this,

Flutter gives you many built-in widgets
that are styled and fine-tuned for iOS.

For example, here in case of the dialogue,

Flutter gives you an alternative show

Cupertino dialogue method.

Now if you want to use this method,

you have to add an import at the top though,

and you have to import cupertino.dart

from the Flutter framework package.

Now Cupertino is simply the name of the iOS styling language

or design language you could say.

So whenever you read Cupertino in a Flutter app,

it refers to iOS styles and behaviors.

So with cupertino.dart imported,

you could call show Cupertino dialogue

to have that Cupertino look and feel.

Now here for this builder,

you then have to pass a function that also takes a context

and that then returns a widget

just as before for the normal show dialogue function,

if you want to call it like this.

Now the widget you would want to return here

would be the CupertinoAlertDialogue.

So here for the normal show dialogue function,

I returned the normal alert dialogue,

which has the material styling.

Here for a showCupertinoDialogue,

I'm returning CupertinoAlertDialogue.

Now, thankfully here for the CupertinoAlertDialogue,

you can also set all these parameters,
which we did set here for the normal alert dialogue,
title, content and actions.

These are all supported here as well
and hence we can just copy and paste them here.

But with that, we have a dialogue
that looks like the iOS dialogue.

But of course, we'll have one problem
we can't show both here,
or at least we probably don't want to show both.

So we need some way of finding out
on which platform we're running
to then show the appropriate dialogue.

So how can we find out on which platform we're running?

Well, for this, we can use an if statement
here nested inside of the outer if statement,
though you could also refactor the code
to be in a separate method
and that is indeed what I'll do here.

I'll add a new void method showDialogue,
and in here, I want to show either the Cupertino
or the normal dialogue.

So I'll cut that code from submit expense data
and paste that in here into show dialogue.

And in submit expense data before returning here,
I'll call showDialogue.

And this is totally optional,
I'm just doing this to keep this method lean and readable.

But now in showDialogue,
we wanna find out on which platform we're running.

And as mentioned, this can be achieved
by using if statement.

And then by using the platform class here,
which is provided by Dart,
which has helpful properties like `isAndroid` or `isIOS`.
And here we can check if the platform is iOS.
And for that, an import must be added at the top
where you import `dart:io`
because that's the Dart package
that provides this platform class.
And with that import added,
we can use this platform class here
and then check if we are on the iOS platform,
in which case, we wanna show the Cupertino dialogue.
So I'll move this code here into this if statement
and else I wanna show the regular dialogue
which I show on Android, therefore.
And with that we have a convenient way
of finding out on which platform we are running
and we can then use this information
to adjust the widgets we're rendering
or like in this case, to adjust the dialogue we're showing.
So here, if a force a reload,
open this and click save expense,
now I have that iOS look.
And that's how you can build adaptive apps
by using this platform class
to determine on which platform you are.
And then you can use that information
to run different code for different platforms.
And, of course, here we're calling
different `showDialogue` methods.
We could also use this platform information
here in the build method

to return different widgets or different widget combinations based on the platform.
All these things are possible.

10. Module Summary

And that's it for this course section.

In this section,
you learned how you can build responsive user interfaces by using MediaQuery to get the width or height of the screen on which the UI is presented,
or by using LayoutBuilder to get information about the constraints applied by the parent widget of the widget in which you're using Layout Builder.

And you can use both the information you get from Media Query or the information you get from Layout Builder to then render your UI or parts of the UI conditionally, to, for example, render a row instead of a column, or vice versa.
You also learned how you can dynamically add extra padding to make sure that the keyboard that might be opened up once the user taps into a text field is respected and the content is pushed up appropriately, and that you might also want to add scrollability to make sure that all input fields are always reachable.
And you also learned about this SafeArea concept, which makes sure that things like the device camera don't overlap your UI.
And of course, last but not least, you also learned how you can find out on which platform you're running

so that you then can execute different code
or show different widgets
based on that platform information.

6_todo_internals

1. Module Introduction

At this point in the course,
we had a look at many crucial Flutter and dart fundamentals.
And of course, we're going to learn about more important
Flutter and dart features throughout the rest of the course.
But before we dive into those other sections
and into more demo apps,
I wanna take a closer look at some important
Flutter internals in this section here,
because understanding at least the basics
of what's happening behind the scenes
of Flutter can help you build better apps
and simply make sure
that you truly understand the technology
with which you are working.
And therefore, in this section, we'll take a look
at the concepts of widget element and render trees
and what these things are and how they are related.
We will take another look at how Flutter updates
the visible user interface,
and how it makes sure that your app stays performant.
And we'll take another look at keys,
which we already used in the previous course sections,
but which might not entirely be clear yet.

2. Three Trees: Widget Tree, Element Tree & Render Tree

So let's get started.
And before we have that look at keys,
let's have a look at trees first.

Because when working with Flutter,
you are essentially working with three trees,
though two of those three trees are typically invisible
to you as a developer.

You do, of course, know the widget tree,
which you're building.

When building a Flutter app,
you're dealing with widgets,
and it's this combination of widgets,
this nesting of widgets into each other,
which we typically call a widget tree.

But Flutter also knows a so-called element tree
and a render tree,
and it helps to understand what these trees are all about
and how they are related.

Now, as mentioned,
the widget tree is simply this combination of widgets
in your code.

So if a Flutter developer takes a look at some Flutter code
and sees all these widgets combined with each other
and nested into each other,
they would typically call that combination
the so-called Widget Tree.

But Flutter actually takes that widget tree
and translates it into the element tree,
which is simply in memory representation of your widgets.

Now, with that, I don't mean the widget objects
that are created by instantiating the widget classes.

So your classes that extend state full
or state less widgets,
but I mean additional objects
which are connected to the widget objects though,

which are created in memory by Flutter.

And Flutter creates these extra element objects in memory to be able to efficiently determine required UI updates.

Now these element objects as mentioned, are related to your widget objects.

For example, when a widget is first encountered and its build method is called for the first time, Flutter will thereafter create a fitting element object and store that in memory.

But the build method is typically called quite frequently, for example, because you called set state in a stateful widget.

In that case, that widgets build method would be called again, and all child widgets build methods would be called again, because Flutter needs to check that entire part of the widget tree to make sure that the state update is represented correctly.

But Flutter does not necessarily recreate all those element objects.

Instead, it only creates new elements if you added new widgets, or it removes elements from the elementary if widgets were removed, for example, because of some conditional code.

So elements are actually reused if possible.

They are not recreated all the time.

They are created initially when a widgets build method is called for the first time, but for subsequent cause of a widget's build method, elements are instead reused.

But it's this element tree that's then used

to determine which UI should actually be rendered on the screen.

And that's where the render tree comes into play, because the render tree is the combination of visible UI building blocks, so the things you can see on the screen.

And creating that render tree and updating the UI and the render tree, therefore, actually takes quite a lot of work.

It has quite a big performance impact, and therefore, Flutter is built such that it really tries to avoid touching that render tree, and it only re-renders parts of the UI when UI updates are needed.

And if an update is needed, is determined with help of that element tree.

Because in the end, you can think of this UI updating process like this.

When the build method is called, Flutter checks the element tree and reuses existing elements as needed, and then thereafter, it compares the new element tree to the old element tree.

And if there are any differences, those differences are applied to the render tree and the UI is updated.

But of course, Flutter will not re-render the entire screen just because one tiny part of it needs to be updated.

Instead, it really takes care to only partially update the UI and re-render the parts that need updating.

Now, as a developer,

you don't need to worry about the element and render tree,
Flutter manages those trees for you.
You build your widgets with their build methods,
and that's it.
But as a developer,
you should always also understand
what's happening behind the scenes,
and therefore, this year is
what's happening behind the scenes.

3. How The UI Is Updated

Now to make this tree thing
and this UI updating process at least a little bit
less abstract,
I created a very basic demo app which you'll find attached.
To be precise, you find my lib folder attached
and you can drag
and drop that into a new Flutter project created
with Flutter Create.
And if you do that and you then start this app
which I provided to you, you should see an app on the screen
that looks something like this.
Now this is of course a very simple not too helpful app,
but it is an app where we do have some interactivity
and it is an app that can help us understand the difference
between our widget tree and the build method being called
and the element tree.
And for that in this ui_updates_demo.dart file,
I created a stateful widget UIUpdatesDemo.
And this is a very simple stateful widget
with a column widget being rendered here,
which displays these texts

and these buttons that change the state.

And then this conditionally shown text
here at the very bottom.

So that's what's being rendered here.

This is the widget tree of this UIUpdatesDemo widget.

And overall this is the widget tree
that's being rendered for my app.

Now if you open the debug console here in Visual Studio Code
and you restart the app, which also clears the console,
you will see two log statements in there, UIUpdatesDemo
CREATEELEMENT called
and UIUpdatesDemo BUILD called.

Now these log statements are coming from my
UIUpdatesDemo widget.

And in there in the build method in its state class here,
you'll find this print statement at the beginning
of the build method, which is of course responsible
for this output here.

The CREATEELEMENT output is coming from the
widget class itself

because there, I actually added a createElement method
which we haven't added before anywhere in the course,
and which you typically also don't need to add.

And here I'm also not doing anything
but calling the parent widgets CREATEELEMENT method
and logging something.

So unless you really know what you're doing, you won't need
to override this createElement method
and implement your own logic.

Though you can do this
because this is indeed a method
that is also invoked by Flutter.

And Flutter calls this method
to create the element for this widget.
And that of course would be the element connected
to your widget object,
and it would be this element
that would be used in conjunction
with all the other widget elements that make up your apps
to determine when the render tree must be updated
and which partial UI updates might be required.
Now what you can of course see here is
that actually `CREATEELEMENT` is called
before `BUILD` is called,
but you can think of that
as the element object being created
and then being updated
behind the scenes once your build method finished.
b=Because it is your build method
that then also tells Flutter, which other widgets
and widget elements must be created
because that's always important.
Flutter does not just create elements for your own widgets,
but for all the widgets that make up the UI.
So the `Padding` widget gets its own element.
The `Center` widget gets its element,
the `Column` widget and so on.
And all these elements form the element tree,
which is then used to find out whether UI updates are needed
and when the app is being rendered for the first time,
of course the entire element tree is new
and therefore the entire UI is rendered for the first time.
So this rendered tree for the UI is created
for the first time,

but thereafter the element tree is only updated if it needs to be updated.

And the same is true for the render tree.

If I for example, clicked this no button now, nothing on the UI changed.

If you take a look at the log though, you see that UIUpdatesDemo BUILD was called again and BUILD was called again

because the build method of a stateful widget, which this build method is in the end here, is always called when the state of that widget is updated, which happens if I click the no or yes buttons because for both buttons I called setState.

Now in case of the no button, I set my isUnderstood state to false though, and that also is my initial state.

So therefore, of course I did update the state and BUILD indeed was called again.

But Flutter then determined that no new elements needed to be created because no new widgets were added.

I have one widget, this text widget here at the bottom which is displayed conditionally, but if isUnderstood as false, it's not shown and therefore its element doesn't need to be created.

So therefore this is a great example for the build method being called and the widget tree being reevaluated and partially recreated, but the element and rendered trees would be left untouched if I click no initially.

Now, if I click yes though, that changes because now this text here appeared and this text is coming from this line of code

where this text widget is rendered conditionally
and therefore if we take a look at the log statements,
if I restart this
and click yes again, we see
that indeed BUILD was called again,
createElement was not called again
because this UIUpdatesDemo widget does not
need to be recreated.
It was part of the element tree before as well.
But indeed createElement
for this text widget would've been called internally.
We're just not logging this here,
but for this text widget,
which now should be shown a new element was created
because initially this text widget did not have a text
element because it was not included
because of this condition.
But once isUnderstood was set to true, which is
what happens when this yes button is clicked,
then this text widget becomes part
of the active widget tree, if you wanna call it like this.
And therefore an element was created.
And therefore, of course, Flutter also determined
behind the scenes that the new element tree differs from the
previous element tree
because it now has this text element, which was not part
of the element tree before.
Therefore, Flutter went to the actual UI to the render tree
and updated it and rendered this text.
Now this does not mean that all the rest
was re-render as well.
Flutter does not do that,

but it did re-render the parts that needed updating.

So this awesome text in this case here.

4. Refactor & Extract Widgets To Avoid Unnecessary Builds

Now with this newly gained logic

about how Flutter manages these different trees

and updates the UI,

you are now also able to write slightly better code.

Because for example,

here in this UIUpdatesDemo widget,

of course, this is not a complex widget;

it's a very simple widget.

And creating this widget like this

would be absolutely fine.

But still,

with our newly gained knowledge,

we could optimize this widget a little bit

because at the moment

when setState is called,

this build method here is re-executed again,

as you learned.

And Flutter essentially checks all these widgets

and their elements,

if anything changed there,

if anything needs updating.

Of course, these checks lead to no updates,

especially if 'No' was pressed,

but still Flutter has to check all these widgets.

It has to execute this entire build method.

Again, for this simple app,

that's not a problem.

But for more complex apps

with more complex widget trees,
you typically wanna avoid checking widgets
for updates that can't possibly be affected
by state updates.

So for example,
here we know that state updates
are really only concerning this text here.

So they are triggered by these buttons
and have an impact on this Text widget,
which might conditionally be shown.

State updates in this app here in this widget
will never impact this Text widget,
this SizedBox,

or this Text widget or this SizedBox
because we're not using state here.

Therefore, we could improve this code
by adding a new widget.

I'll name it DemoButtons here,
since this is really just a dummy app.

And this new widget would now be a StatefulWidget.

So here I have DemoButtons
where I extend StatefulWidget,
and of course,

we have to import material.dart.

And then there as always,

we create our state

and we have our DemoButtonsState class here
where we extend the state

that is based or connected to DemoButtons.

And that's also the case for this state here
in front of the createState method.

And then here as always,

we have our build method,
and now we can go to that other widget file
and grab this Row with the buttons
and this Text widget here
and return that here in this build method.
Though of course,
since these are multiple widgets,
a Row, and then this conditional text,
we actually have to wrap that into a Column
since the button should be shown above this text.
So therefore here we can add a Column widget
with children
where the first child would be this Row
and the second child
would be this unconditionally shown text like this.
And then we have a semicolon,
a comma here for formatting.
And now we would have this part outsourced
into a separate widget and state.
Of course, here for DemoButtons,
we also should add our constructor function
and forward that key.
And in createState
we should return the instance of DemoButtonsState.
And we of course now also need
to manage this _isUnderstood state here.
So we have _isUnderstood,
which initially is false
and which is then set to true or false
when those buttons are tapped.
And now with all of that,
I just want to go to this Column here in DemoButtons

and set the `mainAxisSize` to `MainAxisSize.min`,
which makes sure that this `Column`
does not try to take as much space vertically
as it can get.

So it basically does set a vertical size constraint
to make sure it's only as big as needed
because of the content inside of it.

And on the other hand, in `UIUpdatesDemo`,
we now can include the `DemoButtons` here.
Like this, though for this of course,
we also must add the import,
so this must be added.

Now with all of that,
we can actually turn this entire list here
into a `const` list
because it's full of `const` widgets,
so we don't need `const` in front of all these widgets.

We can just put it in front of the list.

And we of course now can and should convert this back
to a `StatelessWidget`
because now we don't need to manage state in here anymore.

So we can delete all that part
and also delete the `createElement` method,
which was only there for demo purposes anyways,
and convert this to a `StatelessWidget`.

And now we can also grab this print statement
and copy that here into the `DemoButtonsState`
so that we can also see when that is rebuilt.

And with all of that, if we saved it,
restart the app and open the Debug Console.

We see those initial print statements
that the `UIUpdatesDemo` widget is built

and DemoButtons of course is built.

But if I now click 'Yes' here,

you see that indeed for DemoButtons,

we get this print statement,

but we don't get it again for UIUpdatesDemo;

there we still only have

the initial print statement output.

So this proves that

now the build method of UIUpdatesDemo

is not being called again when our state changes.

And therefore this also means

that Flutter does not have to check

all these widget elements

and look for differences or required updates.

And again, for this basic app,

this of course doesn't really matter,

but for more complex apps

and more complex widget trees,

this can improve the performance of your app.

Therefore, it is considered a good practice

to keep your StatefulWidget as small as needed.

Sometimes you need a lot of widgets inside of them

because sometimes a lot of widgets

do interact with state,

but sometimes it's also just a tiny part of the UI

that's really affected by state and changes state.

And in such cases it is a good idea

to put that part into a StatefulWidget

and keep the other widgets out of that widget

so that those don't have to be checked all the time.

5. Understanding Keys - Setup

Okay, so now we learned about widget, element, and render trees.

And especially understanding the relation between widget and element trees is really important.

And it is important to keep in mind that for widget trees, the build method is called very frequently whenever the state changes, for example.

But on the other hand, elements are not recreated unnecessarily.

Now we can also use this newly gained knowledge to take another look at keys.

Because it turns out that in Flutter, this key concept is in the end related to this widget element tree thing and how Flutter updates the UI.

And just in case it's not clear what I mean with keys, I mean these keys which we had to accept as named arguments in all our widgets, and which we forwarded to our parent widgets.

Or the case where we actually used this key parameter to add a key when working with the dismissible widget in the previous course sections.

That's the keys I'm referring to here.

So what's up with all these keys?

For that, I prepared the keys folder with various files, and in there you have this keys.dart file, which in the end defines a to-do class.

And this is in the end a data model, not a widget.

But then I also do have a widget in there, the keys widget, which is a stateful widget,

which then has a bunch of dummy to-do items

that will be output on the screen, and some order state,
some order property that can be changed,
which defines how these to-dos will be ordered.
So in which order they will be displayed.
There then is a gather which uses this order state
in the end to sort this list of to-dos
using a built-in sort method to in the end,
compare to to-do texts with each other,
and compare to we'll do an alphabetic comparison.
And then this sort method in the end
makes sure that the list items are sorted accordingly.
And we use this order state
to determine which items should come first,
depending on whether we wanna sort ascending or descending.
It's also worth noting that sort is being called on a copy
of my to-dos list here.
So that's my dummy to-dos list.
And I'm creating a copy here with list of,
because sort is a method that changes the list
on which it's called.
And I don't wanna change the original list
stored in underscore to-dos,
but instead I just wanna work with a copy
so that the original list always stays the same in memory.
But that's just a side note,
and I'll get back to this concept
of working with data in memory a little bit later anyways.
But this is what I'm doing here.
I'm ordering the to-dos based on my order state.
This order state is changed here in this change order method
and this change order method is simply triggered
whenever this text button here is pressed,

or this icon text button here,
and you will see this button in just a second.
And then down there I have a simple column.
We could also use a list view.
Here, I used the simple column,
because I don't have many to-dos here.
And I output my items here, my to-do items.
With that in list for loop syntax,
which you also saw before in the course,
where I have this for loop without curly braces
to tell Dart which items should be added to this list.
So that's what's happening here.
And here I'm using the ordered to-dos,
so the to-dos that take this order into account.
And we can now use this keys.dart file in the main dart file
by replacing UI updates demo here with keys.
And for this you also must add a import
to package Flutter internals, which is my project name here,
keys, and then keys.dart.
And we can get rid of this import
since we're not using UI updates demo anymore.
So with that, we're using this widget.
And if you now force a reload, you should see this app,
and you can click this button to change the order.
And you see that indeed the order does change.
This item here is always in the middle,
because alphabetically, it's always between the other two,
but the other two items do swap the place,
depending on the order we choose.
And this simple demo app is great
for diving into the concept of keys again.

6. Which Problem Do Keys Solve?

Now to understand what the idea behind keys is and when we need keys, let's come back to this concept of widget and element trees, because you learned that every widget gets its element. That's what we covered in the last lectures.

Now what happens to those elements if two widgets change their place as we're doing it in this Todo app?

There when we reorder our widgets, these widgets change their place. They're not staying in the same position.

So what happens to those elements if those widgets change their place?

Well, the element skeleton actually stays the same.

We keep the same element structure and indeed Flutter does not create new elements here. Instead, it only updates which element refers to which widget.

And Flutter does this for performance reasons.

It simply sees that even though something changed related to the widgets, it's still the same amount of widgets and the same type of widgets.

Because here in this list, when we reorder our Todo's, we might be changing the order of some TodoItems, but we still have the same amount of TodoItems and we also still have the same kind of widget.

We have three TodoItems, even though their text might have changed position.

But for that reason, to optimize performance, Flutter does indeed not throw away those existing TodoItem elements and recreate them, but instead it reuses them.

But it keeps track of which element refers to which widget.

To make sure that in the end, the content
that might be displayed inside of a widget.

So the icon widget

and the text widget that is responsible
for outputting the `TodoItem` specific data,
that those things can be updated on the screen.

To make sure that this is possible,

Flutter connects every element to a widget and
therefore here it reuses the elements,
but updates the references

so that therefore the nested elements that were created
for these `TodoItems`.

So in this case here, the element for the icon widget,
the element for the `SizedBox` widget
and the element for the text widget can be updated
to reflect the correct `TodoItem` data.

That is in the end what happens in this case here,
where some `Todo`'s might be shuffled around.

Now, why is this important to understand?

Because here it explains what happens behind the scenes.

And as you can see here, the app works absolutely correctly
and displays the intended result despite Flutter
reusing those elements.

So we have the best of both worlds.

We have good performance and display the intended result,
but things start to break at least without keys,
if we make one important change inside of key start.

If we replace `TodoItem`,

which is coming from the `TodoItem` Dart file

with the `CheckableTodoItem` widget here

by using `CheckableTodoItem`

and updating the imports to import CheckableTodoItem
and get rid of this import.

If we do that

and we restart the app,

we see that we now can mark these Todo's as completed.

But what you will notice

is that if I only check the first one here

and I then change the order, that looks broken.

The first Todo is still checked.

But of course this technically is now a different Todo.

So now here we have a problem

because even though Flutter does indeed manage element

references, so references to widgets,

and even though it updates those references,

state is managed in a different way.

And here we are dealing with state

because in the CheckableToDo widget,

we indeed have a StatefulWidget.

And there we are managing this Checked State in the end.

And we're getting this broken result here

because these state objects that belong

to widgets are managed as independent objects

that are connected to the elements.

And this actually makes sense

because we also do have separate state classes.

So indeed we also get separate objects in memory.

And before in the course I mentioned

that those state objects would be connected

to the widget objects.

Well, technically that's not entirely correct.

Instead, those state objects are connected

to the element objects that are connected to the widgets.

Now, if widgets change their place,
as mentioned, Flutter reuses elements
and just updates the references, the pointers to widgets,
but the state objects don't move around.
So here you can see that when TodoItem A and B
change the place,
the state's A and B stay in their places.
And that now suddenly connects the TodoItem B
with the State A here
because the state don't move together with the widgets.
Instead, they are connected to the elements,
which also didn't move around, but which instead were reused
and stayed in place
and just got their widget references updated.
And that's of course a problem.
But it is a problem that is solved with help of keys.

7. Understanding & Using Keys

So how can keys help us
solve this problem here?
To understand this,
it's important to understand
that Flutter determines whether an element
can be reused or not
simply by taking a look at the type of that element,
for example, to-do item.
And if that type is in line
with the type it finds in the widget tree
in that position where the element is, it's happy,
and it keeps that element
and just makes sure to update the reference
to the widget it found in the same place

in the widget tree.

That's all.

But it just takes a look at the element type
and the widget type.

And that's why the state doesn't move along
because of course the types here didn't change.

We're still dealing with to-do items,
but this behavior changes
if we introduce keys.

If we add keys to widgets,
those keys are also added to the elements.

And now if your widgets move around,
the keys move with them.

And if Flutter finds a key on an element,
it does not just take a look at the element in widget types,
but also at their keys.

And therefore here it would then see
that the keys don't match.

And it would then also update the key on the element
and move those states.

So it would move the elements, you could say.

And therefore, then, with the usage of keys,
which are simply an additional identification criteria
allowing Flutter to match elements to widgets,

Flutter is able to make sure
that every widget has exactly the element
with exactly the state it needs.

So therefore, here, in this example,
the fix is to go to `keys.dart`

and add a key to our `CheckableTodoItems`.

And that's also why it's recommended
to accept a key in the constructor function

on all your own widget classes

because that makes it very easy to set a key

if you ever need to do that.

Now, the key is then forwarded

to the parent class

so that Flutter can do its thing with it.

But that is why you should always accept this key parameter

so that you can always set it if you need to.

Like here where I actually need to set a key

on the `CheckableTodoItem`,

which also accepts keys, of course.

And we can set that key by adding that key parameter,

which we are accepting in the constructor function

of this widget.

And the only remaining question now is,

which kind of value should we set

as a value for this parameter?

Well, in here, you got two main keys

which you should typically use.

The first one is a so-called `ValueKey`

created with Flutter's `ValueKey` class.

`ValueKey` wants a value, any kind of value,

it's a dynamic type,

so you can pass a number, a text,

or anything like this.

And this will then internally be used

as an identification criteria.

So therefore here we could, for example,

use the `todo.text`

because that should be unique for every to-do.

If our to-dos had IDs,

we might wanna use those IDs instead,

because that would be an even better value.

But here it's the `todo.text`,

and if we add such a `ValueKey` here,

we'll see that now in this app

indeed the state does move together

with the widget.

So now this app is working the way it should.

Now, as an alternative to this `ValueKey`,

we could also use an `ObjectKey`,

and then we could use the entire to-do object as a value.

That is also something you could do,

though I personally prefer the `ValueKey`

simply because, behind the scenes,

it's also a bit more lightweight

because it only needs to manage a single value

instead of an entire object.

But both keys work fine.

The only important thing

is that those values must be unique

and must be directly connected to your data.

So don't use any random numbers

that change all the time

or anything like that,

but instead, use key values

that are directly connected to your data

and that are unique.

8. Mutating Values in Memory & Making Sense of `var`, `final` & `const`

So now we learned a lot about

Flutter's internals.

How it updates the UI and why we sometimes need keys?

Now I wanna conclude this section

by taking a look at a Dart concept
that can be confusing.

In this course,

we sometimes had a case like this

where we had a list, like let's say numbers.

One, two, three

and then we called add

and added a new value to this list.

And this can be confusing because

isn't this a final list,

so why can we add new values if it's final?

And that's a good question

and it's therefore really important

to understand what's going on here.

Final simply means that this variable is final,

so you can't assign a new value to it.

If I would try to set numbers to a different list,

like this, I would get an error

because this variable is final

and it can therefore only be set once.

So put in our words,

I can't use this assignment operator,

this equal sign again on a variable that is final.

This does not work.

This here, however, does not assign a new value

instead it use the existing value

stored in memory.

So this existing list stored in memory

and adds a new value to that.

It does not store a new value in the variable.

The value is the same,

it's the same object in memory, the same list.

It's just that object in memory
that's changed behind the scenes.
When assigning a new value like I tried it to do here,
I'm actually creating a new list in memory.
A totally new object in memory.
I'm not editing an existing object.
I'm creating a new one
and I can't assign this new object to numbers.
That is what final prevents.
And that's what's different if I use var instead.
With the var,
I am allowed to assign a new value.
This is now possible
because with var we create a variable
that can be reassigned.
So where you can overwrite
the existing value with a new value.
You can also still edit an existing value,
but you can also override it.
That's the difference between final and var.
And then we also have const.
With const, you also can't assign a new value,
but here you also get an error
if I open the debug console and restart the app,
if you try to add it an existing value in memory.
Here we get an error that I can't add
to an unmodifiable list.
And I'm getting this error here
because with const,
you're telling Dart that
it's not just final,
but this value

also can't be manipulated behind the scenes essentially.

That's what const does here.

Now, when we used const on widgets,

we're essentially doing the same thing.

We're saying that it can't be manipulated behind the scenes

and that's why Flutter is able to reuse those widgets,

if we use the same widget with the same configuration

in different places of our widget tree

because it knows that it can't change in a magical place

and it therefore can't break one part of the app

just because it changed in another part.

With const, it's clear that this can't happen.

And here we're using const in front of a value,

whereas here we're using it in front of a variable name,

but we could also write this as

final numbers equals const like this

to make this list unmodifiable

and make this variable unmodifiable.

But we can also use a shortcut here

and add constant front of the variable like this.

This makes the value

and the variable unmodifiable.

With final, we make just the variable unmodifiable

and with var, everything is allowed.

So this difference between var, final and const

is really important

and it is important to understand

and keep in mind

that when you create data with var, for example,

this data is stored in memory.

So in this case, this list object is stored in memory.

This line of code which you see here

creates a new list object and stores it in memory
and in the end it's this address of this data in memory
that's stored in the variable.

That is what happens behind the scenes.

Now if you override the value in the variable like this,
a new object is created
and stored in memory with a new address
and this new address overwrites the existing address
in the existing variable.

So in the end, the data in the variable
is overwritten, you could say,
but technically it's the address.

Now the variable points at a different object in memory.

In this case here,
if the old list which only included max
was only stored in users
and not used anywhere else in the code,
Dart would actually recognize
that this old list isn't used anymore
and would indeed remove it from memory
to free up memory again.

But Dart does this automatically.

This is a process called garbage collecting.

So that is what happens if you create new values
and store these values in variables.

Now for many objects,
you cannot just create those objects,
you can for example, not just create a list,
but you can also manipulate existing objects.

For example, by calling add on a list.

And this indeed does not create a new list,
but it adds the existing list in memory.

The address is the same because the object is the same.
It's just some internal data of that object that changed
and therefore the address and the object stays unchanged.
It's still the same object in memory
and therefore this is a use case
where we could also use final instead of var.
Now speaking of final, as mentioned,
if you're using final,
you're not allowed to reassign the variable
because that is what final prevents.
It makes sure that variables are final.
They can't get new addresses that they should store.
They can't store new values, therefore,
but you can absolutely add existing values in memory.
This is allowed with final.
And with that, this final, const, var thing
hopefully makes a bit more sense
and is a bit less confusing
when encountered in code.
This is by the way, also,
why in keystore when getting these ordered to-do's,
I'm creating a copy of my to-do's
with the built-in of method
that exists on the list class,
which is spilled into Dart,
so that I don't change the original list
because sort like add is a method
that does change the list on which it is called.
But I instead only change the copy
because I don't want to change the original list here.
But that's just a side note.

9. Module Summary

And that's it for this course section.

In this section, we learned how Flutter manages different trees to update the UI efficiently, and that, for example, the element tree tries to reuse as many elements as needed.

And the rendered tree is only updated partially if differences between the old and the new element tree are detected.

And you also learned why you need keys when working with stateful widgets, which otherwise might lose their state if used without keys because the state would not move along with the widgets.

With keys, it does, and that's why sometimes in your Flutter apps, you need to add keys to make sure that they work correctly.

And last but not least, we had another look at var, final, and const and what the differences are and what the difference is between assigning a value and editing a value in memory.

7_meals-app_multi-screen-n-navigating-app

1. Module Introduction

It's time for another app.
And in this course section,
we're going to build this meals app,
which allows users to browse various meal categories,
pick meals and read the meal ingredients and instructions,
mark meals as favorites,
switch between all meals and favorited meals,
and also switch to a totally different page
where various filters can be set to filter the meals
that will actually be shown in this all meals area here.
That's the app we're building.
And the special thing about this app is
that it has multiple screens
between which we the user can navigate.
We can go forth and back
and use the tabs to go to different areas of the app.
We can use the side drawer to switch to a different screen,
and that is of course, something we haven't done at all
in this course up to this point.
But this navigation between different screens is
of course something you will need
in a lot of apps you might be building.
And therefore, this course section here is
about navigation in Flutter apps,
and how you can build multi-screen apps
that do allow users to navigate between those screens.
And for that, we're going to learn

about the concept of screen stacks
and how we can dive deeper into an app by adding screens to
that stack or going back by popping screens off the stack.
You will learn how to add a tab bar
and how to make sure that users can use that tab bar
to switch between different screens.
And you will also learn how to implement a side draw
and how to use that to switch between different app areas.
And as always along the way,
you will also have plenty of opportunities
to practice what you already learned up to this point,
to work with more widgets,
and to also learn about some new concepts and widgets.

2. Project Setup

Now for this section here,
I got another new starting project
created with Flutter Create.
I named it Meals.
And in this project, I got a main.dart file,
which you'll also find attached to this lecture here.
Now, this main.dart file
has a couple of issues right now.
We're going to fix them in a couple of seconds.
But you should use that attached main.dart file
and replace yours with it.
And once you replaced your file with it,
you should install the google_fonts package
since I'm using the google_fonts
to actually overwrite the default text theme
with one of the themes defined by that package
that uses a different font family,

the Lato font family, in this case.

This is something we haven't done

up to this point in this course,

but it's a very convenient and easy way

of overriding the default text theme

with another text theme

that does use this specific font.

And the `google_fonts` package

offers pre-built text themes

for all the fonts it provides.

Therefore, you should install

this `google_fonts` package for Flutter

by following the official installation instructions,

which simply means that

you should run this command here.

And once you did that,

the first error will have been fixed.

`google_fonts` can be found now.

And the only remaining error now

is that you must set some widget

as the main entry widget of your app.

3. Using a GridView

So we need a main entry widget for our app.

And for this app I'd like to get started

with this Categories screen,

where we can then select a category

and see all the meals that belong to that category.

And therefore, I'll add a new file here

in this project lib folder,

which I'll name `categories.dart`,

because in there I wanna build the widget

that displays all these categories.

Now inside of categories.dart,

you should therefore create a new class.

The CategoriesScreen class, for example.

The name is up to you.

And you could also just name it Categories.

But since this will act as a screen in this app,

and we will later have multiple screens,

but also some regular widgets which don't act as screens,

I wanna mark the widgets that will act as screens.

So therefore, this one is the CategoriesScreen.

And we of course extend StatelessWidget here.

And here we'll only need a StatelessWidget,

because indeed, there won't be any internal state

that would need to be managed in that widget.

Sure, we'll be able to select a category later,

but as you will see, the navigation to the screen

that shows all the meals for the category,

will not be handled with state,

but instead with a different mechanism.

And therefore, this indeed can and should be

a StatelessWidget.

Now of course, as always, we need the build method,

and we should add a constructor function here,

CategoriesScreen is a fitting name,

where we should accept and forward

the key property as always.

And here in the build method,

I wanna return the Scaffold widget

because I wanna have that default styling that it applies.

I want to be able to add the AppBar at the top and so on.

And in general, it's very likely in Flutter

that if you're building a multi-screen app,
that every screen will use the Scaffold widget
to do its base screen setup.

Because for example, the AppBar often differs
between different screens.

Different screens might have different titles
or buttons in the AppBar.

And therefore, being able to set the AppBar
or other screen-specific settings
right from inside the screen widget,
is a common requirement.

And hence, it is quite common to set up the Scaffold widget
inside of the screen widget, as we're doing it here.

So here, I will indeed add the AppBar parameter
and set this to AppBar.

So, I'm using the built-in AppBar widget again,
as we did it before in the course.

And on that AppBar,
I'll set the title parameter to the Text widget
and output some text as a title in this AppBar.

And the text here could be Pick your category,
or anything like that.

Now we can also add const here,
since this is a constant widget.

But of course, that's not all I want to do
in this Scaffold here.

Instead, we also wanna set the body.

And the body, as you learned before,
is the main page content.

Now here for the body,
I wanna output all these different categories,
which then later can be selected

to move to one of those categories.

And for this app here,

I plan to lay out these categories as a grid.

And thankfully, creating such a grid

is also quite straightforward in Flutter,

because it does not just offer a ListView

for rendering a list, but also a GridView for,

you guessed it, rendering a grid of items.

And here, just as with the ListView,

you also have a builder constructor function,

which would allow you to build the grid dynamically

if you had a potentially very long list of grid items.

But alternatively,

you can also instantiate the grid view widget like this,

to not get this possible performance optimization

of only rendering what's visible.

But here, I won't have that many categories,

so that performance optimization

wouldn't really kick in anyways.

And therefore I'll use this approach,

where I can directly set my children.

So, the items that should be displayed in the grid.

Now, GridView also needs something else.

As you can see in this error.

It needs a gridDelegate, which should be set.

And therefore, besides setting the children,

which we will soon do,

you also must add the gridDelegate parameter here.

And this is a parameter which might sound quite strange,

but in the end, if you hover over it,

you learn that this in the end simply controls the layout

of the GridView items.

And here, I'll create a grid
with this `SliverGridDelegateWithFixedCrossAxisCount` option,
which is a very complex way of saying that
I can, in the end, set the number of columns
which I want to have here.

So here, I can set my `CrossAxisCount` to 2.

And since by default, the `GridView` `mainAxis`
is from top to bottom, the `crossAxis` is from left to right.

So if I set this to 2, I get two columns next to each other.

So, horizontally, I have these two columns.

Now with that set here, we can also set a `childAspectRatio`,
which will impact the sizing of those `GridView` items.

And here, I'll set this two 3 divided by 2.

Two in the end, dynamically derive a three to two ratio.

Alternatively, you could also manually set this to 1.5.

I'm just using this notation to make it clear
that this in the end is the aspect ratio
which I want to have.

And I'll also set a `crossAxisSpacing` of 20,
to have some spacing between the columns,
and a `mainAxisSpacing` of 20 as well,
so that I have a spacing of 20 pixels horizontally
between my items, and vertically between my grid items.

And this can then also be turned into a `const` object.

So, that's the grid configuration.

Now we can add our grid items here.

And I'll add the actual data soon enough.

For the moment, just to see whether it works,

I'll just add a bunch of dummy text widgets,

1, 2, 3, 4, 5, and maybe 6.

We can add `const` here

to see whether this grid is displayed as intended.

So with that all set up,
we can use our Categories screen in the main.dart file
here as a value for home,
where we can now use CategoriesScreen like this,
and actually add const front of it.
And of course, also make sure that you add ed the import
to the categories.dart file.
With that added, and this set here, you can save this.
And if you then run the app on an emulator,
as we did it many times before in the course,
you should see a screen that looks something like this.
And of course, the text is not super-readable here.
So let me quickly fix this,
even though this is just a temporary placeholder
and will soon be removed anyways.
But here, I'll quickly add the Style parameter
to my text items to set some color,
and manually set this to Colors.white,
because again, this will be gone soon anyways.
And then we can copy this to all these Text widgets here,
just so to make sure that the text is a bit easier to see.
And we can indeed see that we now get this grid.
The styling definitely isn't finished,
but we can see the two-column layout
along which the different grid items are positioned.
So that's good. Now it's time for the actual category data.

4. Widgets vs Screens

So the goal now
is to replace these dummy text widgets
with the actual category data
and widgets that present that category data.

And for that, first of all, in the lib folder,
I'll add a new models folder
because I'll add a new file that contains the blueprint
for the category objects with which will work.
So I'll name this file in there category.dart.
And in this category.dart file,
I'll add a new class, category, which is not a widget,
but just a blueprint for a regular object.
And here, I'll define how a category
in this app should look like,
how it should be structured.
And of course there are different ways of defining this,
but for this app here, a category should have an ID,
which should be a string.
It should have a title, which is also a string,
and it should have a color, which is of type color.
So that's a Flutter-specific type
and therefore we should import material.dart
by importing package:flutter/material.dart
as we did it many times before.
That's how a category should in general be structured.
And of course, I'll also add a constructor function here,
a const constructor function, which will be used
to initialize these objects and set these properties.
So in here, I expect to get an ID.
I also expect to get a title and I want to have a color,
but there I'll actually set a default color of color orange
like this so that if no color would be provided,
this fallback default color would be used.
So that's how for this app, a category should be defined.
And with that defined,
we can add some dummy data.

For this, also in the lib folder, I'll add a data folder
and in there I'll add a new file,
which I'll name dummy_data.dart.
And in this file, I wanna store some dummy categories,
which we'll use for this demo app and attached,
you find my dummy.dart file
that contains some predefined categories.

So here I got a bunch of categories
and we must add an import to our category model
to make this work,
though the attached file already includes this import.
There, you just have to make sure
that the project name in this import path
matches your project name.

But with that, we're importing the category model
and we're setting up all these categories.

And we also need to import
the material.dart file from Flutter, but again,
this will also be already the case
if you're using the attached dummy.dart file.

And in there, in this dummy.dart file,
you see that a bunch of categories are created
all with IDs and titles and colors.

And it's these categories here
that should now be output on the categories screen
that is defined in the categories.dart file.

Now to output these categories in this categories screen,
I'll add another new folder in my lib folder
next to data and models, which I'll name widgets.

And the idea here is that I put all non-screen widgets
into this widgets folder,
and therefore I'll actually create another folder

on the same level as data models and widgets,
which I'll name screens
because all my screen widgets should now go in there.
So I'll move the categories.dart file
into the screens folder.
And in the widgets folder,
I'll create a new category_grid_item.dart file.
And this is simply a file that will contain a widget,
which we'll then use in this grid for a single category
so that we don't end up with one huge widget here,
but we instead outsource the category grid item logic
into a separate widget.
So with that in the category_grid_item.dart file,
we have to create a new class for this CategoryGridItem,
so for this widget,
and this does not have a screen
because it won't act as a screen,
but it should of course extend stateful or stateless widget.
And indeed here, we will have a widget that will not need
to manage any internal state.
And therefore, this can and should be a stateless widget.
And of course, you must add
that material.dart import as always.
And we have to add this build method.
And also, as always, of course,
our constructor function here.
Now we should accept and forward the key parameter.
But in addition, this class here,
this widget class, should also accept some external data
that will be required
to output the actual category information.
Indeed, in here, I simply expect to get the category

for which I wanna display information
as an input of type category.
And therefore you must make sure
that you add the correct import to your own project package.
And then there are two models, category.dart,
so that this category type is available
because the category class is defined in that file.
And of course then here, I also expect to get this data
as an input.
Here, I'll expect it as a named parameter,
but as always, you could also go for
a positional parameter instead.
But here, I'll go for a required named parameter,
which also has the name category
and is then automatically set to this property.
And with that, we can return our widget combination
for this category grid item widget,
a combination that should simply display the category title,
use its color,
and also make sure that the item will be tapable
so that later, we can navigate to a different screen
because ultimately that's what this section is about.

5. Displaying Category Items on a Screen

So, here I'll start by returning a Container.
And I'm using Container because the Container widget
gives me a lot of options for setting the background color
or background decoration in general for this widget.
And that is indeed something I'll need to do here.
I wanna set the decoration parameter
that is supported by container
and set it to a BoxDecoration object

created with help of the `BoxDecoration` class,
which is provided by Flutter.

And I'm using this class here because with that
I can create a gradient as we did it before in this course.
And therefore, as always, also feel free to pause this video
or also the other videos in this section
to sometimes try out the next steps on your own.

Because here, my goal now is to create a linear gradient
that uses the category color
to transition from one variation of that color,
which is simply the color with some opacity
to a different variation.

So, the same color without or with less opacity.

And I'll do this here by creating a `LinearGradient`
and setting the `colors` parameter to a list of colors.

And now, my first color will be my `category.color`
with some opacity.

Here, I'll go for an opacity of `.55`
and my target color will be the same color,
but with less opacity.

So, that we'll get a nice subtle gradient effect in the end.

And I'll have a target opacity of `.9` here.

Now, we can also configure this gradient
and define where it begins and ends as you learned.

And here I'll make it begin by setting `Alignment.topLeft`
and set end to `Alignment.bottomRight`.

And of course, I also need to add a comma here.

So with that, I'm adding a decoration to this `Container`,
but that's not all.

The `Container` also should get a child.

Hence, we should set the `child` parameter on the `Container`.

And the child is simply the widget

that will be placed inside of the Container.

And here I wanna have the text widget

with the category.title as a text,

but also with some styling applied.

And here for the styling, I'll use my theme with Theme.of,

as you learned it in the Themeing Course Section,

to reach out to the textTheme.

And then, there the titleLarge theme,

I wanna apply the styling of the titleLarge theme

in the end.

But I'll then copy this with some extra settings.

Though for that, I must add an exclamation mark here

to tell Dart that the titleLarge text theme

will be defined, which it will be,

since I'm using this GoogleFonts Define TextTheme here,

which does a Define TextThemes

for all the different text variations.

So, this will work.

But with that, I can copy this default TextTheme.

And then, here I just wanna change the color

and set that to Theme.of(context).colorScheme

to access the color scheme that was inferred by Flutter

based on some base color that I set in main.dart.

This base color here.

And then, from that colorScheme,

here I'll choose the onBackground color.

And you know it.

You can and should always play around with things like this

to try out different colors or different configurations

so that you really get a good feeling for how Flutter works

and how different widgets and objects

can be configured in different ways.

So, that's the color I wanna apply here.

And with that added, there's just one last thing.

For now at least.

I wanna add some padding to this container.

And the offer, I'll set the padding parameter,
which thankfully is supported by Container,
which makes Container a quite versatile widget.

And I'll set the padding to `EdgeInsets.all(16)`
to add a padding of 16 pixels in all directions.

And this is then a const object.

With that, we can go to `categories.dart`,
because there I wanna use this `CategoryGrid` item,
which I just created.

And we should delete those dummy text widgets now.

And instead use the dummy categories that are defined
in the `dummy_data` file and output those here
in the Categories screen in the `categories.dart` file.

For this in this list here,
which is set as a value for children,
we can use a for in loop to go through all the categories
in available categories
which is the name of this constant value here
in that `dummy_data`.

And for that to work in this screen file,
you of course, must import the `dummy_data.dart` file,
which is in the data folder into this file here.

So, this import here must be added.

But with it added, you can go through all the categories
and we have to remove const now,
since now we're deriving the list contents dynamically.

And when going through the categories here,
you can then define the widget

that should be output for every category.

And here, that will be the `CategoryGridItem`.

And of course, `CategoryGridItem` is only available if you add this import here.

Where you import the `category_grid_item.dart` file, which is stored in the `widgets` folder into this file here.

So with that imported, we can use this here.

And it is also worth noting that using this for loop here is only an alternative to using `availableCategories.map` and then getting hold of every category like this and returning the `CategoryGridItem` like this in here, and then calling `toList`.

So, that would be the alternative to writing it like this.

I mentioned this before, but it's really important to keep things like this in mind, which is why I'm mentioning it again here.

So with that, we're outputting one `CategoryGridItem` per category in `availableCategories`, and we're passing the category as a value for the category parameter to this widget.

And with that, if you then save this and force a reload, you should see these category items here.

This is not entirely the final look I wanna have, but it's a good start.

This grid view is also scrollable by default, by the way, if you would exceed the available screen space here.

But with that, we're almost there.

I now just wanna go back to the `GridView` and also set the padding parameter on that, which thankfully is supported to also give this `GridView` some padding.

And here I'll choose `EdgeInsets.all(24)` as a value

so that these items don't sit directly on the edges.

And with that, this doesn't look too bad.

We got these nice categories here.

But of course, now I wanna make sure

that A, they have rounded corners,

and B, more important, they are tappable.

Because right now, if I tap them, nothing happens.

And that, of course, needs to change.

6. Making any Widget Tappable with InkWell

So Let's now make sure

that these grid items are tappable.

And for that I'm going back to my category

grid item dart file and in there we can make this container

or any other widget tappable by wrapping this widget

that should be tappable with another widget

and that other widget would be the InkWell widget.

Now actually, the InkWell widget

is not just about making some child widget tappable

that could be achieved

with the GestureDetector widget as well.

If you wrap a widget like this container

with GestureDetector, you can add a broad variety

of listeners to it.

For example, onTap but also may other listeners

to listen all kinds of user interactions

with the child widget.

But if you use InkWell instead, you can also add

an onTap or many other listeners as you can see,

but in addition you get a nice visual feedback

if a user taps the item.

With GestureDetector you would get no feedback,

with InkWell you do, that's why I'm using InkWell.

With InkWell wrapped around container,

I can add the onTap parameter to set up a function

that will be triggered whenever this item will be tapped.

So with that, we already would be able to listen to events,

but I'm not done yet.

In addition, I'll set a splash color which will be important

for this visual tapping effect that we get.

And I'll set this to theme.of(context).primaryColor

to use one of my theme colors for that

or as a basis for this effect to be precise.

And we can also set a BorderRadius here

which does not want a number, but instead a BorderRadius

object which can be constructed in different ways.

For example, by using the circular constructor

which allows us to define a radius like 16

to give these items some nice rounded corners.

You can try out different values here

to get exactly the look you want.

So with that these items should be tappable.

Right now nothing happens when we tap them,

but they are tappable.

And if we saved this all and force a reload

we can see this effect in a very subtle way,

if you tap and hold one of these items,

you see they light up a little bit,

but of course we don't have the rounded corners

or at least we don't have the rounded corners

on these items right now.

If we tap one and hold this,

the highlighted fact indeed has rounded corners.

So the missing thing here indeed only is

to go to this container in our category grid item
and there in decoration, besides adding this gradient,
I also wanna add a BorderRadius
and just as on the InkWell this BorderRadius
should be set to BorderRadius.circular(16), like this.
And with that, now we finally get these rounded corners
and this nice highlight effect if we tap and hold.
Of course, the next goal is to make sure
that we cannot just tap and hold,
but that we actually navigate to a different screen,
if we tap one of these items.

7. Adding Meals Data

So the goal is to go to a different screen
if one of these items is tapped
and therefore we should add a different screen.
In the screens folder,
I'll create a second file, which I'll name meals.dart,
because the goal with this screen is to show all the meals
for a selected category.
Now inside of meals.dart,
we of course must create a new class,
and I'll name it MealsScreen
because it will be a widget that acts as a screen.
And this will also extend StatelessWidget
because there also won't be any state that would need
to be managed in this screen later on.
So MealsScreen like this, it is.
And of course we add build
and we add our constructor function as we always do
for wording that key.
And for this screen, I will also need some input

because I will need a list of all the meals
that should be displayed for a given category.
And for that as a first step, we of course need meals
because otherwise it'll be hard to display them.
So therefore, back here in `dummy_data.dart`,
I want to add some dummy meal data.
And for that, in turn,
I first of all want to add a new model
in that `models` folder.
I'll add a new file called `meal.dart`,
in which I wanna define a structure for a single meal.
So in there I'll add a new `Meal` class.
And since this would be a rather long class,
you find my `meal.dart` file attached.
And you should simply replace your `meal.dart` file
in the `models` folder with the attached one,
or simply replace the content.
And if you did that, you will see
that my `meal.dart` file contains this `Meal` class,
which actually has quite a long list of properties
that are all set through named parameters.
For example, for every meal we store a title, an image URL,
the ingredients, which is a list of strings and the steps.
So basically the instructions on how to make that meal,
which is also a list of strings.
The duration, which is a number, and then also complexity
and affordability, which are enums
with some predefined choices
or some predefined options to put it like this.
In addition, I also have some flags here
that basically tell us whether a certain meal
is gluten-free, lactose-free, vegan, or vegetarian.

So that's how a meal will be defined in this app.

And with that model added,

we'll also update the `dummy_data.dart` file

and you can simply use the one which you find attached

and replace your `dummy_data.dart` file with the attached one.

And then again,

make sure that your project name matches

here in the import statements.

And then you'll see that we still have

all these categories here.

But then also this list of dummy meals

where I use this newly added `Meal` model to create a bunch

of dummy meals, all with ingredients, instructions,

and some links to images.

Though of course, definitely feel free

to set up your own meals here in addition.

So with those meals added to the `dummy_data.dart` file,

we can go to the `meals.dart` file.

And in there we now wanna output that list of meals.

Now since this widget here will act as a screen widget,

we also first of all should return a scaffold widget

so that we can, for example, set the `AppBar`,

which is exactly what I'll do here

so that I can set the title.

And the title here should actually be set

to the category that was selected.

Therefore, in this `MealsScreen`,

we should accept that title as input data.

And I also wanna accept my meals as input data.

And that should be a list of meals

using that meal model that we defined in the `models` folder.

Hence this import path here must be added.

And of course, these two input values now also should be set when this widget is created.

So here I will require a title to be provided

and I'll require the meals to be provided.

And back down here in the build method,

we can now set the title, well, as a title in the AppBar.

And set the body to this list in the end.

So to a widget that outputs this list of meals.

And that is of course also something

you could try on your own.

Try outputting this list of meals here,

maybe in a way that offers good performance,

even if we had a long list of meals.

And also make sure to show some fallback text

in case no meals should be available.

So if we selected a category that has no meals, for example.

Definitely try this on your own.

Thereafter, we'll implement this together.

8. Loading Meals Data Into a Screen

So, did you succeed?

Let's now build this together.

And for this here in this scaffold body,

I'll use Flutter's `ListView.builder` constructor here

to create a scrollable list view that makes sure

that only items that are actually visible will be displayed

to optimize performance for scenarios

where we might have very long lists.

So here we then must pass a value for `itemBuilder`,

and that, as you learn, should be a function

that takes context and an index.

And then this simply builds all these list items

and for every list item that gets built here,
I want to output the meal information.
In this case here for the moment,
simply a text widget that outputs the name of the meal.
For this, I'll acts as my meals list,
and then one specific meal by index
and then simply the title like this.
Now of course, we also could have a scenario
where we got no meals and therefore to handle this as well,
I will actually add a if statement here to my build method
where I check if meals.isEmpty
so if I get an empty list of meals here.
And in this case,
I wanna set my body to something different.
Therefore, I'll add a new variable of type widget,
which defines my content or body, however you wanna name it,
I'll go for content
and here I'll set this to this list view here like this.
But if my meals are empty,
I'll instead set content equal to the center widget
where I then output a child.
And here my child will be a column
where I set the main axis size to min
so that this column really is centered
and is not unconstrained vertically.
And the children of the column here
are two text widgets on the end.
The first text widget simply says, "Uh-oh, nothing here."
And then I have a sized box for spacing
with a height of 16, and these two widgets here
can be turned into constant objects.
And then I want to have another text widget

where I could say, "Try selecting a different category."

And here, I now actually wanna apply a specific style with Theme.of, and then I'll access the textTheme and use the bodyLarge text variant to then copy it with that exclamation mark in front of it to tell Dart that this text variation will be defined in this theme.

So to copy it with a different color, that's the goal here in the end.

I'll use a different color and the color I want to use here will also be taken from the theme and there from the colorScheme and it should be the onBackground color.

And to make sure that I get some auto formatting here and this becomes readable again,

I'll quickly set this content as a value for body down here in the scaffold so that I got no more errors in this code file.

And now after formatting the code, this is in the end how my content looks like if my meals are empty.

I'm outputting this text here.

And I indeed also wanna apply since I'm styling to the first text, so I'll copy over this style parameter with its value, but here, I will access headlineLarge, so a different predefined text variant, but then also set this text color to the onBackground color.

And this now can't be const anymore since I'm deriving some theming data dynamically.

Now of course, these exact styles, as always, are up to you.

And we could also switch the order here by the way, and set this here as our default value for content

and set this list view here

as a value if meals is not empty, so if we have meals.

But ultimately this is up to you.

I'll go for this default here though,

and I'll put my content

and now to see whether or not that worked,

I'll go to the main.dart file

and temporarily I'll replace the categories screen there

with the MealsScreen.

And of course for that, you must import the meals screen,

the meals.dart file from the screens folder.

And then on that screen,

you of course have to pass a value for title,

like for example, some category.

And soon this will be dynamic,

but for the moment, all just hard code some dummy data.

And then the meals, which we could set to an empty list

to see whether our fallback content is displayed correctly.

So let's see whether that works

if I restart this, looks good,

or to show the actual meals that we have

and for that, it's worth noting that in dummy data,

the meals list has a name of dummy meals.

So in the end, it's this list,

which I wanna use in main.dart now, dummyMeals.

And for this to work, you must import

the dummy_data.dart file into this main.dart file,

but with this set, we should be able to output the list,

though I am getting an error here because indeed,

I forgot in the meals.dart file on this list view

to tell Flutter how many items I'll have and therefore,

I'm getting an error.

Indeed here, we should set the itemCount
and set it to meals.length.
So this line of code here is important to make sure
that Flutter renders the list view correctly.
With that, it works and the styling is off,
but it's also not the final styling,
but we can see some meal titles here.
A bit hard to read, but again, this is not final yet.
We'll, of course, improve this, but the general logic works.

9. Adding Cross-Screen Navigation

So at this point we can output
this basic unstyled and unfinished meals list here,
this meals screen to be precise.
But of course at the moment this is all hardcoded.
In main Dart, I always output the meals screen
with all my dummy meals and a hardcoded title.
That's not the goal.
Instead, the goal is to, here, output the CategoriesScreen,
as we did it before,
and to then make sure that if we tap a category,
we're navigated to that category's meals screen,
which then only shows the meals for that category.
And that's therefore what we're going to implement now.
For that, as a first step, we can get rid of
the meals screen import here in the main Dart file,
and of the dummy data import there.
And instead go to the categories.dart file,
where we have this categories screen.
And there, I want to add a new method,
and I'll name this method _selectCategory.
And it won't return any data, hence its return type is void.

And here, we're doing something
we haven't really done before.
We are adding a method to a StatelessWidget.
Now, we haven't done this before
because in most cases when we added a method to a widget,
so when we reacted to a button tap, for example,
we wanted to update some state.
And of course we can't do that in a StatelessWidget.
But here, we don't want to update state.
Instead, I want to load a different screen.
And for this, we can use a feature built into Flutter.
We can use the Navigator class,
which you already saw earlier in the course.
Earlier in the course, we used it with its pop method
to remove a screen and go back to another screen,
even though you didn't know
that we were kind of working with screens back then.
But this modal, which we for example closed,
in the end also was kind of like a screen that was opened.
And with Navigator.pop, we were able to close it.
Now we don't want to close a screen,
but instead we want to navigate to a screen.
And this can be done by using the push method.
Now the push method wants a context value
and then a route value.
And in the end, what push will do
is it will push this route,
which in the end kind of will be a widget,
on top of the current stack of screens.
And this can be a bit of a strange term
when you hear it for the first time.
But in the end, in Flutter,

navigation basically works with screen stacks.

You can think of the screens being organized as layers put on top of each other.

And as a user, you always see the topmost layer, the topmost screen.

So, that's why it's called push.

You push a new layer onto this stack of screens, which you already have.

And on the other hand, if you pop a screen, you remove one of these layers.

And since the user always sees the topmost layer, the user then sees the screen below the screen that was popped off.

That's the idea here.

So here we're pushing, and we need to pass in a context and a route, as you learned.

Now, since I'm in a StatelessWidget here, context is not globally available.

You might recall that in the state objects that belonged to Stateful widgets, a context property was globally available, and could therefore be used in all the methods.

Here, that's not the case.

Therefore here, we indeed must accept context value, which must be of type BuildContext in this select category function here, so that we can pass context as a value to push.

By the way, alternatively, you can also call Navigator.of(context).push, and then only pass the route to push.

This is in the end, the same as writing this code like this, and it is up to you which approach you prefer.

I'm just putting this out there because you'll see this kind of code in a lot of Flutter projects as well. But that's how you then push a new screen onto this stack of screens. But what's up with this route now? Well, such a route is created by instantiating a `MaterialPageRoute`. Som by creating a `MaterialPageRoute` object with help of the `MaterialPageRoute` class, which is provided by Flutter. And this gives us such a route that can be pushed onto this stack. And as you can see, since I used autocompletion here, this wants a builder parameter, which wants a function that gives us a context. So, we will get a context automatically in this builder function and we should return a widget as part of this function. And that widget will be that screen widget that should be displayed. So here, I have a function that automatically receives some context object, and in there I then return my `MealsScreen`, in this case. Now for this to work, we also must import this `MealsScreen`. So here, from the screens folder, I'll import `meals.dart`, so that `MealsScreen` is available here. And now of course, we still must provide a title and then our meals. That hasn't changed. So, for the moment I'll just say Some title and pass an empty list of meals. Though this will change later.

But for the moment, that's it.

And this is how we would push this meals screen
on top of this stack of pages.

And since it would therefore be the topmost screen,
it would be the screen users see.

But of course, right now,
selectCategory never gets executed.

So, we have to make sure that it does get executed
by connecting it to our CategoryGridItem.

Or to be precise, by connecting it
to this onTap parameter here,
inside of our CategoryGridItem widget.

Well, and you learned how that works,
inside of CategoryGridItem,
you can simply expect to get a function,
which could be called onSelectCategory.

And this then, should be received as a value, like this.

And if you do accept this parameter,
and bind it to this property inside of the categories screen
where we are outputting these category grid items,
we can now set onSelectCategory.

And here for example, set it to an anonymous function
where we then execute selectCategory.

And here we then pass our context to selectCategory,
so that inside of selectCategory we have this context.

With that done here,

if you now go back to the CategoryGridItem,
and also make sure that onSelectCategory is connected
to this onTap parameter.

So, passed as a value to onTap so that this function,
which we just edit here,
is indeed executed whenever this InkWell is tapped.

So in the end, whenever this `CategoryGridItem` here is tapped, if you do wire it up like this, you can save everything and force a reload.

And now if you go back and you tap a category, you see that we get a nice little animation, and the screen changes.

And we get this back arrow here automatically added by Flutter.

We did not add this manually to the app bar. It was added by Flutter.

And if we tap this back arrow, it will automatically pop that screen off this stack of screens.

So that we get this expected behavior of being able to go back, and Flutter will manage this screen stack for us behind the scenes.

Of course, at the moment, we always see the same output for all categories because we're not actually loading category-specific data.

But this is one way of navigating between different screens in Flutter, and it's crucial to be aware of this feature.

Well, and I guess, now you are.

10. Passing Data to the Target Screen

So, now that we can go to the Meals screen, it's of course, time to populate it with the actual data for the selected meal.

And therefore, here in `selectCategory`, we should make sure that we load the meals that belong to the selected category.

For this as a first step,

we should accept a second parameter here,
which is the category that was selected.
Because without this information it will be quite difficult
to find the meals that belong to a certain category.
So, therefore here, we should make sure
that we actually import this category model
from the category.dart file in the models folder,
and we accept this parameter.
And then here in selectCategory,
we should reach out to all our dummyMeals
and use the built-in where method,
which is supported on all lists in Dart
to filter this dummyMeals list.
Because where essentially returns a new Iterable,
but that, as you learn, basically is a list
that only contains the items that match a certain condition.
So, to where, you pass a function
which will be executed automatically by Dart
on every item in the list on which where is called,
so on dummyMeals.
So here, we get the individual meal items,
because this function will be executed once per meal
in that dummyMeals list.
And we should return true if we wanna keep that meal
or false if we don't wanna keep it
in that newly created list that will be returned by where.
Now, to define a fitting condition here,
let's take a brief look at the meals again.
And here, it's important to note
that every meal has a categories property,
which is a list of strings,
and in the end, it's a list of category IDs.

So, the categories have IDs like c4, c5, and so on,
and in categories in our meals, we have those IDs,
and every meal has different category IDs
to which it is mapped.

So, therefore here, in the categories screen
where I wanna filter my meals,
I can take a look at my meal.categories
and check whether it contains,
which is never built-in method available on lists
to check whether a list contains a certain value.

And here, I'll check whether meal.categories contains
the category.id of the category that was selected here.

And that will return true if this meal
for which this function is currently executed
does include that category.id,
and false otherwise.

And therefore, the list returned by where
will only include meals
that do have this category.id added to them.

So here, I'll then call to list
to convert this Iterable return by where to a real list,
and store it in a variable filteredMeals,
or however you wanna call it.

And in the end, it's now this filteredMeals list
that's passed as a value
to the meals parameter on the MealsScreen.

And the title, of course, can now also be set dynamically,
and it should be set to the category title
so we can use this category parameter down here
to access category.title.

And with that, we should be able to load the actual category
and meals data for the category that was selected.

Of course, now we just must make sure
that this category parameter
is also provided when selectCategory is executed.
So, therefore, down here
where we're calling selectCategory,
besides passing in the context,
we now also must pass in the category,
but thankfully, this category is available here
in this loop here.
It's available there,
so therefore, we can set or pass this category here
as a value to selectCategory.
If you now save this and force a reload,
if you click a category,
you can see the title indeed was used here.
And even though it's hard to see,
the meals here are now also filtered,
and we don't have all the meals being output here.
And different categories show different titles
and a different list of meals.
And of course, with that,
it's now also finally time to fix the way
these meals are displayed here,
and make sure that they are actually displayed
how they should be displayed.

11. Introducing the Stack Widget

So let's now work on these meal items here
and let's make sure that
those look the way they should look.
For that, I'll start by adding a new widget,
so not a screen widget but a regular widget, so to say,

that will be included in another widget
instead of being loaded as a screen.
And I'll name this widget MealItem
or I'll name the file MealItem.dart,
but in there I'll name the widget class mealItem as well.
Now of course, as always, we must extend
stateless or stateful widget,
and here again it is stateless widget.
We will also use stateful widgets later in this section.
But here this widget is really just about
outputting the meal item data in that list of meals
which we're displaying on the screen here,
so this is just a stateless widget.
And then of course here we'll need to get some data
and store some data.
Most importantly, we'll need the meal.
So you should import that meal model file
so that you can store the meal as a property here,
and of course also get that meal data
as input to this widget, so as a parameter.
So besides getting and forwarding that key,
I'm also requiring this meal parameter which then is stored
or which value is then stored in this property here.
And then of course, as always, you need the build method
where you return the widget combination for this widget.
And here for this widget,
which will be output in a list in the end,
I'll return the card widget,
which is built into flutter
and which you saw before in the course
in the expense tracker app already.
And here in this card, I then will output a child,

which will be another InkWell widget
because these meals should also be tapable,
because of course later we wanna be able to select meals
from that list of meals here
and then be taken to the meals detail screen.

That's why I'm using InkWell here.

So here we can already set the onTap parameter
on this InkWell widget constructor function,
though at the moment it's just an empty function,
later this will change of course.

And then here we also need another child
inside of that InkWell,
which should actually be a stack widget.

And that's a new widget which we haven't seen before.

It's not related to this stack of screens,
which I mentioned earlier,
instead, this is a widget that can be used to
position multiple widgets above each other
but not above each other as we do it in a column
where they are rendered from top to bottom on a screen,
but instead directly above each other
so that we can, for example, set a image as a background
and have some text on top of it.

That's the goal here

and that is what we can do with help of the stack widget
that's built into flutter.

A stack-like column or row wants multiple child widgets
and hence accepts a children parameter
because we will stack multiple widgets on top of each other,
otherwise we wouldn't need to use stack.

And here we then add all the widgets
that should be positioned on top of each other,

starting with the one that's on the bottom of the stack.

So that's in the background, so to say.

And in my case here, this will be an image

which I will display with help of the

FadeInImage widget provided by Flutter,

which is simply a utility widget

that displays an image that's being faded in

so that when the image is loaded it's not popping in,

which can be quite ugly but it's smoothly faded in.

For that, you also need to define a placeholder

that's shown initially.

And here I'll use another package,

Flutter transparent image,

which simply gives us a dummy image

which we can use as a placeholder,

which will be transparent,

so where we won't see anything

but which will ensure that this fade in animation

between nothing and the final image is played smoothly.

So therefore here we should install this

transparent image package by executing this command here.

And with that being installed as a placeholder,

we wanna display a memory image,

which is another class provided by Flutter,

a so-called image provider class,

which knows how to load images from memory,

and to memory image we pass kTransparentImage,

which is imported from that TransparentImage package,

which we just added.

So we have to add this import

where we import transparent image dart

from that transparent image package.

And with this import added,
this global variable here will be available
and this in the end will simply contain such a memory image
that is loaded as a placeholder.

And then the final image
that should be loaded will be a network image,
so an image fetched from the internet.

And we need that because in our dummy data,
these meals actually have image URLs
which are URLs to images that are stored in the internet.

So therefore I wanna load the images from these URLs
or with help of these URLs,
and `NetworkImage` is exactly the kind of class
that allows us to do that.

`NetworkImage` wants such a URL
and therefore here we can use that meal
which will be stored in this property
to access the image URL on that meal.

And that will make sure that ultimately this image here
will be loaded, but it will be loaded in a very smooth way,
it will be faded in once it's available
after displaying nothing as a placeholder.

And of course you could also display some other image
as a placeholder in the beginning
but here I'll go for this solution.

So that's the first widget in this stack of widgets.

The next and indeed already final widget here
should then be a box, a container on top of this image
that displays the name of the meal
and also some meta information about the meal.

And for positioning a widget on top of another widget
inside of a stack,

Flutter gives us never very useful widget.

And that's the positioned widget.

Positioned takes a child like for example here a container, and it then also allows us to set some additional parameters besides setting this child.

For example, we can set top or bottom or left to define how this child widget should be positioned on top of this widget.

So for example here, if I set this to bottom zero, left zero, and right zero like this, this container will be positioned on top of this image.

This will always be the case because it's a stack and this widget comes after this widget.

And I mentioned before that the first widget will be the bottommost widget and other widgets will be added on top of it.

But with these coordinates I'm making sure that this container will be added on top of the bottom of this image and it will span from the very left to the very right so that it starts at the beginning of the left border of this image and it ends at the right border.

So there is no distance between the left border and the right border and this container, that's what this says.

If I would set this to right 50, for example, I would say that this container should end 50 pixels before this image here ends to the right, that's how you can think of that.

And it's best if you simply play around with these values to then see a difference.

But of course to see a difference,

we first of all have to make sure
that we do display something in that container here.
And for that, in that container, I wanna add a color,
a background color to it.
And here I'll use a built-in color with colors,
and then black54, which is a transparent black color.
So it's not fully opaque,
instead it has some transparency added to it.
And we could also tweak this transparency here
if you wanted to,
but I'll stick to the built-in one.
And with that we'll have a slightly transparent
black background, which I do set up here
to make sure that my meal title and the metadata
is always readable.
Because of course the background images
could be very different
and therefore I wouldn't know how to color the text
that sits on top of them
whilst ensuring that it's always readable,
and that's why I'm using this container,
which always assigns this
slightly transparent black background to my texts.
And speaking of that,
of course we also need to add the text.
By adding a child here, which then holds a column,
because I want to have my title sit above my metadata.
So I want to have two rows with the first row being my title
and the second row being the metadata.
I also wanna add some padding before working on that column.
And here I'll set the padding for this container
to EdgeInsets.symmetric to have different padding

to the left and right and top and bottom.

And here I'll simply set the vertical padding

to let's say five or six, doesn't matter,

and the horizontal padding to 44.

But these values of course are up to you.

And now here in this column I'll set up my children

where as mentioned, the first row will be the title.

And of course here we'll use the actual meal title.

And the second row then should be some metadata.

So therefore I'll actually add a sized box

between these two rows to have some spacing

with a height of 12, let's say.

And then below that I have a row with more children

where I have all these metadata items for my meals.

But I'll get back to those in a second.

First of all, let's output the title.

And for this we can of course refer to meal title.

And then I'll also restrict this

to be no more than two lines long,

in case I would've a very long title,

it would then be cut off.

And I wanna ensure that the text is aligned in the center,

which can be done with help of the text align parameter.

So by setting text align here to text align center,

I ensure that the text is centered.

In addition, I'll add soft wrap through

to make sure that the text is wrapped

in a good looking way, if it should need wrapping.

And I'll add the overflow parameter,

which controls how the text will be cut off

if it needs to be cut off because it's too long,

for example, because it would take more than two lines.

And here we can set this to various options
that are predefined on the `TextOverflow` enum,
and here I'll set it to `ellipses`,
which makes sure that if we had a very long text,
it would simply be cut off
by adding three dots after it,
which looks better than suddenly cutting it off like this.

Last but not least, I'll add some styling here.

And here I'll define a brand new style.

I'll not use my theme here,
instead I'll define a style where I set the font size to 20,
the font weight to `FontWeight bold`
and the color to white like this.

And of course here we can then add constant front of this
and that's how the title should be styled
and configured for this meal item.

Now with the title added, we can work on the metadata,
but before we do that, let's see whether this all
looks and works the way it should.

So to test this,
we can go to our `meals.dart` file in the `screens` folder,
which renders this list of meals.
And here, instead of outputting this dummy text per meal,
we now wanna output the meal item.

And of course you must add the import
to this newly added `MealItem` dart file
in the `widgets` folder therefore.

And with that added we have to pass in the meal
that is being rendered here,
and for that, since we're in `ListView` builder,
we access `meals`
and then we get the index of the meal here as our input,

and that gives us a single meal,
which is then passed as a value
to this meal parameter on the meal item clause here.
And if I save this now and force a reload
and I pick Italian,
I now indeed see my meal here.
Now the image first needs to load initially,
and as you can see, it does fade in smoothly,
but then you see the meal here
and you see the title below that.
Now I'd actually like to have some spacing
around these meals,
and for that we can go to MealItem again.
And on this card we can add a margin,
it's a parameter which conveniently is supported
by the card widget.
And we can set this margin here to EdgeInsets.all
and maybe add eight pixels in all directions.
And in addition,
I'll actually also change the shape a little bit
and set the shape to a rounded rectangle border,
which then allows me to give this card rounded corners
or change the border radius
with the border radius of my choice.
And for that here
in this rounded rectangle border clause constructor,
we can set the border radius parameter
and set this to border radius.circular
as we did it before in other places of the course as well,
and maybe choose a radius of eight.
As always, definitely play around with these settings
to get a better feeling for that.

Now if I do that and save everything,
you will see that I don't really have rounded corners.
And the reason for that is that this stack here by default
ignores the shape I might have set up here on this card.
To enforce this shape here, I have to add a setting
that ensures that any content
that would go out of this shape is simply cut off.
And this can be done
by setting the clip behavior on the card
to clip `hardEdge`.
And this simply clips this widget,
removing any content of child widgets
that would go outside of the shape boundaries
of this card widget.
Last but not least, I'll add the elevation parameter
to add a slight drop shadow behind the card
and give this card some elevation,
some 3D effect therefore,
and this is simply a int number
where again you can play around with different values,
I will simply use a value of two here.
With that, you can save this.
And now you can see that if you reload this,
we do have the rounded corners here.
Last but not least, on this fade in image,
I'll also add the fit parameter
and set this to `BoxFit cover`
to make sure that the image is never distorted,
but instead cut off and zoomed in a bit
if it wouldn't fit into the box otherwise.
In addition, I'll add a fixed height of let's say, 200
to make sure that we don't have this ugly jump

which we otherwise could have initially
before the image is there.
And now to make sure that the entire width is used,
I'll set the width here to double infinity,
to tell the image to use
as much space as it can get horizontally.
And thanks to this setting here,
it will use this height and this width
without being distorted.
So with that, this now looks much better.

12. Improving the MealItem Widget

So we're making good progress here,
but right now the metadata is missing.
The metadata which I wanted to display
below all these new titles.
And for this, I have this row here,
but of course at the moment, this row is empty.
Now, my goal is to output the duration, the complexity,
and the affordability of every meal in that row.
And if you take a look at the dummy data again,
this is all data we are storing in there,
affordability, complexity, and duration.
Affordability and complexity are enum values,
and duration is simply an integer.
So we have this data, we now just need to output it here.
Now, for every metadata item in this row here,
I plan to have an icon with some text next to it.
And since I will repeat that same combination of icon
and text three times, it makes sense
to create a custom widget here,
which we can repeat,

instead of copy and pasting a bunch of code.

And therefore I'll create a new widget here.

`meal_item_trait.dart` is the file name.

And in there I have my `MealItemTrait` class,

which extends `Stateless Widget`

because we won't need to manage any state in here.

And we should import `material.dart` here.

And then as always, we have the build method.

And then our constructor function, `MealItemTrait`,

and we'll get some input data here.

To be precise,

I expect to get the icon that should be displayed,

which should be of type `icon data`

because that is the kind of data the builtin icon

which it wants later on.

And I want to get a string,

the label that should be shown next to the icon.

And here in the build method, I then want to output a row

which has this icon as a first widget,

then a sized box for some spacing,

and then a text widget that displays the label.

Now for that, of course, in this constructor,

we should accept and forward the key parameter,

but then also require and set the icon

and the label like this.

And with this all accepted and set,

we can use it down here.

For example, with the built-in icon widget,

this as mentioned wants an icon data value

as a first positional argument.

So therefore here I can use this icon,

which we have as a property

and which we're getting as a parameter, as a value here.

And I'll actually also give this icon a fixed size

by setting the size parameter that's accepted

by the built-in icon widget.

And set this to 17.

And I'll set the color of the icon to white like this.

Now, for the sized box, I simply wanna have some width

between icon and text, so some spacing,

and I'll set this to a width of six.

And for the text, this of course should output the label.

But here I'll then also add some styling,

some text style, where I simply wanna ensure

that we always have some white texture

so that this label is always white.

And this can be turned into a constant object here.

And that's already all.

This is my MealItemTrait widget,

which I can now reuse instead of the MealItem Dart file.

Here in this row at the bottom,

I'll add a first meal item trait.

And of course, as always for that,

you should make sure that you import

this MealItemTrait.dart file in which we just worked.

And then here we can set an icon and then a label.

Now, the first piece of metadata

which I wanna output here is the duration.

And for that, I think the schedule icon is quite good,

because that's this little clock icon here.

Well, and then the label is the meal duration,

but that, as mentioned before, is an integer.

And of course, label wants a string,

and therefore we have to convert this to a string

either by calling to a string,
or what I'll actually do here by wrapping this
with single quotes,
and then using this injection syntax here
about which you learned earlier,
which also allows me to then add a blank
and then "min" for minutes thereafter,
to make this a bit more meaningful
and useful to the users of our app.
So that's my first MealItemTrait being used here.
If I save this and reload,
you therefore see this item here below the title.
And you see that for all the meals, of course.
Now, by the way, you might be wondering why here we can use
a row, which my MealItemTrait returns in the end,
inside of another row.
Normally we would get an error
and we would need to use Expanded, right?
Well, here it works without Expanded
because this row is inside of a column which is inside
of this positioned widget,
which has left and right set to zero.
Therefore, this positioned widget enforces its child,
the container, to take exactly that width
between the left border of this stack
and the right border of this stack.
And therefore we actually have a fixed width.
And this fixed width is passed down from the container
to the column and from the column to this row.
So therefore, this row here
is not unconstrained horizontally.
And hence it has no problem with another row being

inside of it because even though this row here
is unconstrained horizontally,
there is a maximum width applied
by this row to this MealItemTrait here.
That's how this works here.
So that's the first piece of metadata being output here.
Then I'll add a sized box here for some spacing
with a width of let's say 12.
And then the next MealItemTrait.
But here, I'm not outputting the duration,
but instead the complexity.
And I think here the work item is a good choice.
Now, here I therefore want to output the complexity,
but the complexity is an enum.
And if we take a look at the dummy data,
we see that it, for example, is a complexity of simple.
Now, I'm fine with using this as a text,
but I wanna make sure that it starts
with an uppercase character
instead of a lowercase character.
And to ensure that this is the case,
we need to transform this text here.
And for that, I'll add a simple getter
to my meal item class here,
a getter that will return a string
and a getter which I'll name Complexity Text,
without these parentheses here.
And in there we can now return meal.complexity.name,
because name gives you access to the enum value as a string.
And I'll access the first character
by using square brackets zero here,
which is a notation you know from lists.

This accesses the first list item.

Well, turns out that you can use the same notation on string to access the different characters.

So this is used to access the first character in a string, because just as in lists, the character indexes start at zero.

And now we can call the built-in two-uppercase method on this first character to transform it to uppercase.

But of course, now we would return only the first character in uppercase.

And that's of course not what I want.

I still wanna return the entire word in the end.

So therefore, we have to combine this uppercase first character with the rest of the word.

And we can do this with the plus operator.

You know this operator from mathematical operations, but it can indeed also be used with strings to concatenate two strings.

For example, "Hello" plus "world" would yield "Hello world" like this.

So therefore here I want to concatenate this uppercase character with the rest of this word.

So I'll again access meal.complexity name.

And now to get the word without the first character, we can use the built-in substring method, which is a method that's built into Dart and available on all strings.

And substring simply returns a part of a string starting at start and potentially ending at end where start and end are string indexes.

So here, since strings start at index zero,

I pass one as a value because I don't want
to start at the first character
but at the second character,
since the first character is converted to uppercase here,
and I then wanna append the rest of the word
starting at the second character.

So that's how we can transform this Complexity Text.
And then we can use this Complexity Text down there.
So, instead of accessing meal.complexity,
I access Complexity Text.

And now we can actually get rid of these curly braces here,
since we now have a simple variable
or property that we're accessing here
instead of a more complex expression.

Well, and now we can repeat that,
and again, add a sized box with the width set to 12.

And then one last MealItemTrait,
where I'll choose an icon of attach_money,
which is this icon here,
to output the affordability of the meal.

And this is also an enum
and therefore should be transformed in the same way.

So therefore, I'll just copy this code from there
and create a new getter called Affordability Text.

And here I access meal.affordability.

And I also do this here.

And with that we got this Affordability Text,
which now is output in that last MealItemTrait item.

And of course, I always wanna get rid of min here,
because we don't need that,
and therefore we can actually get rid
of this entire injection syntax,

because we don't need it anymore,
because I just wanna output the complexity
or affordability texts.
Well, and with that, finally, if we save this,
you will see that if you restart this,
now you got all these metadata items
below the title of the meal.
Now, as a last step, I wanna make sure
that this metadata is centered,
because right now it's clearly not centered below the title.
And to achieve this in my metadata row here,
I'll add a main axis alignment
of Main Axis Alignment Center.
The main axis in a row is from left to right,
so this will center the items horizontally.
And with that, this now looks much better.
So now it's time to make sure
that we can also tap those items
and then navigate to the meal detail screen.

13. Adding Navigation to the MealDetails Screen

So now that we do have these meal items here,
it's time to make sure that we can also go
to a Meal Detail page,
so that we can tap such a meal, and be taken there.
And this is a great practice for you
because it's essentially a similar process
to what we did to get from Categories to this list of meals.
I want you to add the code to this project
that allows users to tap those meal items
and gets them to a Meal Details screen.
Of course, you have yet to add this Meal Details screen,

and for the moment it doesn't have to be too fancy.

Just make sure that you output the meal title
in the app bar, and then below that, the meal image.

And we'll then work on the rest
of this Details Screen together thereafter.

But first, give it a try
on your own. (paper crinkling)

So, did you succeed?

Let's build this together,
and let's make sure that users can tap those meal items
to be taken to the Meal Details screen.

Now for this as mentioned,
we first of all have to add such a Meal Details screen,
which is what I'm doing here,
adding a new file in that screens folder
called meal_details.

And then in there, we have the MealDetails class.

And just as before here,
we can extend StatelessWidget like this,
because indeed here we won't need to manage any state.

And we need our build method,
and then of course also our constructor MealDetails
like this, where we get and forward the key.

And for the moment, this will be a StatelessWidget.

Later we will turn this into a StatefulWidget,
but for the moment we just need a StatelessWidget.

Now here we'll also need to store some data.

We'll need to get some data to output those meal details.

And to be precise, we of course need the Meal,
so the Meal which should be of type Meal.

And for that of course you must import the meal model file.

Then of course we also have to make sure

that we get this meal parameter

and store the value received on that parameter

in this Meal property.

Now that we get that meal, we can return our Scaffold here

because we have a brand new screen, as mentioned.

And in this Scaffold I want to have an AppBar

using Flutter's AppBar widget,

where the title is set to a Text widget,

where I output the title of the meal that was selected.

So of this meal here.

Now, besides the AppBar,

we of course also have a body in that Scaffold.

And for the moment, as mentioned,

it's enough if we simply output the image of that meal here.

For that, we can use Flutter's Image widget.

And as you learned earlier in the course,

this widget has various constructor functions

that allow us to load images from different sources.

And here I need Image.network,

since as mentioned earlier, our images are in the end URLs,

two images that are stored somewhere in the internet.

An Image.network loads these images from those places

and then displays them in this app.

So here we got Image.network,

and then I can access meal.imageUrl here

to set the URL for this image.

Now in addition, I wanna assign a width of double.infinity

to make sure that we take all the available space,

and I'll give this a height of let's say 300,

and then add fit BoxFit.cover

to make sure that the image is sized appropriately,

but not distorted.

And with that, we're outputting this image here.

The rest will follow later.

But with that, we got the MealDetailsScreen,

as it should actually be called,

since we'll use this widget as a screen.

So to be consistent with my other widget names,

this should be MealDetailsScreen.

And with that added here, we can go back

to the meals.dart file where we have the MealsScreen,

which displays all the meals,

and there we wanna make sure

that if a single MealItem here is tapped,

we go to the MealDetailsScreen.

Now for that of course, inside of that MealItem widget,

we have our InkWell, which allows us to register a function

for the tap event.

So whenever a user taps on this item.

Now here in the end, in this MealItem widget,

I don't wanna handle the tap,

but instead I just wanna get a Function

as an additional property.

So a Function here, which gets the Meal

on which I tapped as an input,

and which will be stored in a property

called onSelectMeal, for example.

And of course we then must make sure

that this property is set

when the class is instantiated like this,

and it's then onSelectMeal

that should be connected to the InkWell.

So that's set as a value for the onTap parameter.

This however doesn't work

because onTap does not provide such a Meal.

And onSelectMeal is a function that wants to get a Meal,
and therefore here I'll set onTap
equal to an anonymous function.

And in there we can manually execute onSelectMeal
and then pass our meal as a value to onSelectMeal.

And this meal, which I am passing a value here,
is this Meal property.

With that, in the meals.dart file,
where I'm outputting these MealItems,
I now have to set this onSelectMeal parameter,
which we just added, to the MealItem widget.

So therefore here in meals.dart,
I want to add a new function,
which could be named selectMeal.

And this should get the selected Meal as an input.

And that's in line with the parameter
we just added in MealItem,
because that wants a function that gets a Meal as an input,
and we're now creating such a function
in the meals.dart file.

This function here, it takes a Meal as an input.

And in there I then again want to use the Navigator
built into Flutter, of context,
and therefore will have to make sure
that I actually also get my BuildContext here like this.

And then we can again call push here,
to push a new screen onto our stack of screens.

And this new screen here is the MealDetails screen.

Though as you learned, not passed into push like this,
but instead wrapped in such a MaterialPageRoute,
which has this builder where we get the context,

and where we then return our MealDetailsScreen like this.

And now it's selectMeal, which I in the end,
want to use down here as a value for onSelectMeal,
though it is worth noting that selectMeal
now of course is a function that does not just get a Meal,
but also a context.

So now we could either add it meal_item.dart
and make sure that here we also accept a context
in our type definition.

And then we would need to pass a context down here
when we call onSelectMeal,
or alternatively in meals.dart,
we don't set selectMeal as a value for onSelectMeal,
but instead we set another function as a value here,
which gets a meal, because inside of MealItem
where we are calling this function,
in the end we do pass such a meal.

But then here we manually call selectMeal
and we kind of forward that meal that we're getting here,
but we're also passing the context to selectMeal.

And the context here is available
since we're inside of the build method
where we do have this context.

And that's how I will do it here.

So we call selectMeal here
in this place inside of meals.dart,
forwarding that meal which we get from inside MealItem.

And then in selectMeal we are pushing our MealDetailsScreen
as a route onto our stack of screens.

Well, and that should be all that needs to be done.

If we now save this and force a reload,
we should be able to select a category and then a meal.

And as you can see,
I am taken to that newly added Meal Details screen.
And of course we still automatically get this back button,
which we can then use to go back
and pop screens off that stack of screens.
But that's how this forward backward navigation works
in the end, simply by using Navigator push.
And if we ever would want to pop a screen off manually,
for example, because some button was clicked,
we could of course always do that
by using Navigator of context pop,
or alternatively as we did it before,
Navigator pop and passing the context to it.
We don't need to do this here,
but that is how you could manually go back
to the previous screen.

14. Improving the MealDetails Screen

So now that we got the basic navigation
set up, it's time to work on this meal details screen
because of course that screen is not finished yet.
At the moment, I am only outputting my image
and my title here, and that's not enough.
Instead, this image here should just be the top most element
on this screen right below the title and the app bar.
But below the image, I then want to have a list
of ingredients and a list of step-by-step instructions.
And therefore,
here in the meal details screen widget I'll put this
image here into a column widget since we now won't just have
one single widget here as a body, but multiple widgets
and the image is just one of them.

So I'll cut that and paste it in here.

And now below that image, I want to have my ingredients.

Now for that, I'll start by adding a sized box

to have some spacing between the image and the next section.

And I'll give this a height of let's say 14.

And then I'll add a text widget, which kind of acts

as a title for the next section of this screen.

And the next section will be my list of ingredients.

Hence here, I'll give this a title of ingredients.

And this can simply be hardcoded like this.

Now I wanna adjust the styling,

and for that I'll set the style to a style

that's taken from my theme.

So I'll access theme of context, text theme,

and here I want to access the title, large style.

Now I will add this exclamation mark to tell dart

that this will exist, so

that then I can call copy width on that title,

large text variant.

And now this also can't be const anymore, by the way.

So now that I copy this, I then wanna overwrite the color

and set this equal to theme

of context color scheme like this.

And there choose on background as a color.

So that's now the title for the next section.

Below that title, I then wanna output all the ingredients

of the selected meal.

And if you take another look at the dummy data, you will see

that the ingredients is near just a list of strings, just

as the steps will be a list of strings thereafter.

So therefore here we can create a for loop

to loop through all the ingredients.

So for final ingredient in meal

dot ingredients.

This is how we can go through all the ingredients here.

And I'm using this for loop syntax that can be used
inside of a list.

This list here to add items to it.

And the item I wanna add

for every ingredient here is a text widget

where I output the ingredient,

because again, meal ingredients is a list of strings.

So here I'm outputting a string.

If I save this

and I select a meal, you now see the title ingredients here.

And below that, some ingredients, which

of course are not readable though.

So in order to make this readable,

I'll copy this style set up here from my ingredients title

and add this here to this text.

But here, I don't want to use tidal large as a base style,

but instead body medium.

And of course you can play around with different styles here

to find the one you personally like most.

Now I think some spacing between this ingredients title

and the actual ingredients would also be nice to have.

Therefore, before this for loop, I'll add a sized box

with a height of 14 maybe,

so that we have some spacing here.

And I'd say that looks quite decent

and I think it might look even a bit better if we go back

to this title here and change the color from on background

to primary

so that the title is highlighted like this.

We can even make it bold by also setting font weight
to font weight bold here
to make the title stand out a bit more.
And with that, I'm outputting my ingredients.
Now, in addition to the ingredients,
I of course also wanna output the step-by-step instructions,
and therefore we can again, add a sized box below this
for loop here and here, maybe add a space of 24 pixels
before we then copy and paste this title.
We could also put it into a separate widget,
but here, since it's not a lot of code,
and I'm only repeating it once,
I'm just copy and pasting it.
So I'm outputting it here.
And then I'll say steps, and then I'll copy
and paste this loop again.
We could put it into a separate widget,
but here I'll just copy and paste it.
And actually I'll copy
and paste it together with this sized box, which we added
between title and loop.
To add this here below this steps title.
And here, of course, I don't go through meal ingredients,
but through meal steps.
And here I have my individual step,
which is then output like this.
If we now save this, we got the steps here.
And this of course looks ugly and is not that easy to read.
So here I definitely wanna fine tune this
to improve the looks of that.
I'll go to that text
and add a text alignment of text align center.

So that's not inside of the style,
but added directly to the text widget.
And an additional, I'll wrap this text widget
with some padding,
some symmetric padding though where
horizontally I want to have a padding of let's say 12.
But vertically I want have a padding of 16.
And now I'm getting a warning here
because now I'm actually exceeding the available screen
height, but I'll fix this in a second.
If we take a look at it, this looks better though.
Now it's actually too much space.
So let's actually switch from vertical 16 to vertical eight.
And nonetheless, I get this warning,
but now I think this looks a bit better.
Now, to fix the warning, we have to make sure
that this entire screen here is scrollable.
And to make sure that it's scrollable,
we can wrap this column here with single child scroll view,
as we did before in the course,
or we replace it with list view, not list view builder,
because here I actually won't have a super long list
of items, but list view to get this automatic scrolling.
So now with that, you see it is scrollable.
Now my titles
and the ingredients are pushed all the way
to the left though,
because unlike column, they're not centered by default.
And therefore here, I'll instead go back to column
and wrap this with another widget with
that single child scroll view widget I mentioned
before, since this will make sure

that the content stays centered.
And with that here, I can now scroll
and my content is centered, which I would say looks better.
And with that, we finished this details screen.
Now we're actually going to come back
to it a little bit later
and add more features to it
because you should be able to mark meals as favorites.
And this is a feature which we will add later,
but for the moment it's finished
and we can now dive into another navigation pattern.
And that would be tabs, and having a tab bar at the bottom.

15. Adding Tab-based Navigation

So now that we've got these
forward backward navigation working
and we can view our meal details,
it's time to add another
very important navigation pattern to this app,
a pattern which you will likely need
in many of the apps you're going to build in the future.
And that pattern would be tab based navigation,
having a tab bar at the bottom,
which can be used to switch between different screens.
That's what we're going to add here,
and we're going to add it
to allow users to switch
between this categories screen here,
where they then can dive into Categories and Meals
and a Favorite screen
where they can view all their favorite meals
and then still dive into those meals,

but they don't have the categories,
they just have their favorite meals there.
That's the idea and that's what we're going to build here.
Now for that, we first of all have to add a new screen,
a new file in the screens folder,
and that new screen will be called tabs.
Because this tabs navigation requires its own screen,
which then loads other screens as embedded screens,
you could say.
Now in this tabs.dart file,
we therefore create a new class
because we'll still have a widget,
but here we have to extend StatefulWidget
for the first time in this section,
because switching between these tabs,
which we'll soon add,
will require us to update some state
to update what's visible on the screen,
because if we switch from tab A to tab B,
we of course must update the screen.
So here I create a new StatefulWidget Class,
and of course we'll also create
the state class that belongs to it,
where we extend state that is connected
to our TabsScreen widget.
And in this TabsScreen widget,
we have the create state method where we return
an instance of our TabsScreenState like this.
And here we can change this to state of TabsScreen.
We will also add a constructor here,
which in this case is a very simple constructor
that only accepts (indistinct) as key.

And then in the `TabScreenState` class down there,
we of course need to add the `build` method as we always do.
And in this `build` method here,
we will return a `Scaffold` with an `AppBar`,
which is set to the `AppBar` widget
where I want to set the title dynamically.
It should always be a `Text` widget,
but the text that is displayed
as a title should depend on which tab is selected.
So I'll just put `dynamic` here and we'll replace this later.
Now of course, we'll also have a body,
and this also will need to be set dynamically
because that will be the active screen,
which will depend on which tab is selected.
So here, I'll just add three dots for now,
and we're going to work on this later as well,
because now we're also going to add something new.
We're going to add the `bottomNavigationBar` parameter
to the `Scaffold` widget
because this tab bar is such a common thing,
such a common requirement in mobile apps
that it's officially supported by Flutter.
The `scaffold` widget allows you
to set a bottom navigation bar,
which the end simply is such a tab bar.
Now here, `bottomNavigationBar` wants a widget,
but typically here you create a `BottomNavigationBar` widget,
and that's another widget built into Flutter.
Now, `BottomNavigationBar` is a widget that, for example,
has a `onTap` parameter, which wants a function
that will be triggered automatically
whenever the user selects a tab.

So whenever the user taps one of the items in this tab bar.

Now for this `onTap`, we'll call a function that automatically receives an integer as a value, the index of the tab that was tapped to be precise.

So here we should provide a function that accepts this index.

And again, the index value will be provided by Flutter since it will be Flutter that calls this function when the user taps one of our tabs.

And then we also have to add items to this `BottomNavigationBar`, which is a list of tabs in the end.

Now a tab here is a bottom navigation bar item object, and those objects can be created with help of the built-in `BottomNavigationBarItem` class.

Now every `BottomNavigationBarItem` needs an icon.

And in addition, you can also set a label.

Now here I'll need two tabs, one for the categories screen and one for the favorites screen, which we have yet to add.

Now here I'll start with the icon for the categories screen.

You can of course use any icon of your choice, but here I'll build one with the built-in icon widget because icon wants a widget, but of course it typically wants an icon widget.

And the icon I want to use here with this Icon widget is the `set_meal` icon, which is built-in to Flutter and which looks like this.

Now the label here should just be a string.

And here I'll use a label of Categories because with this tab,

the user will go to the categories screen.

Now for the second BottomNavigationBarItem,

I'll create a icon with the star icon,

which looks like this,

because here we'll go to the favorites

that were selected by the user.

Now here we can then add const in front of this list.

Since both items are const

and can't change in any way,

we can, of course, change which one was selected,

but the tab widgets themselves internally don't change.

And now we can set our body

and of course that body should be set dynamically

based on which tab was pressed.

So therefore what I'll do here in my TabsScreenState class

is I'll add a new variable,

which is of type int which I'll name selectedIndex.

And initially I'll set that to zero.

Now I'll have two pages

and I simply plan to work with the index

as zero for the first page

and one for the second page.

Now with that state added,

I can also add a method that can be called selectPage,

where I get an index as a input

and where I then call setState

to update my selectedIndex

and set it to the index I get as an argument here

and it's now selectPage,

which I'll connect to my bottomNavigationBar

as a value for onTap

so that whenever we tap one of these tabs,

we trigger `selectPage`

and we update the `selectedPageIndex`.

Now, how is this page index useful?

Well, here in the build method,

we can now create a new variable `activePage`,

which is of type `Widget`.

And by default,

this could be the `CategoriesScreen` like this.

And we can set this to `const` here,

but now I can check if the `selectedPageIndex` is one,

so if the second tab was selected.

And keep in mind

the second tab should lead us to the Favorites.

So if this index is one,

then I wanna set `activePage` to a different screen.

And indeed the screen to which I wanna set it

is the `Meals` screen.

Now, we used the `Meals` screen before

to get there from inside the `Categories` screen,

and that won't change.

We will still push this `Meals` screen from inside there.

But the Favorites screen can simply reuse the `Meals` screen

because on the Favorites screen,

I in the end just wanna display a bunch of meals

and that is exactly what the `Meals` screen does.

So reusing that screen makes a lot of sense.

Therefore, if the `selectedPageIndex` is one,

so if this second tab, this Favorites tab was pressed,

my `activePage` should be the `MealsScreen`.

And of course for `CategoriesScreen`

and `MealsScreen` to be available in this file,

you have to add imports to both screens.

So from the screens folder in your project,
you have to import categories.dart and meals.dart.
But with that imported,
you can set these screens as values for activePage.
And now for the MealsScreen,
we also have to set the title and the meals
and the title here should be Favorites,
though I will come back to that a little bit later.
But for the moment it's Favorites.
And for the meals,
we of course wanna show just the favorite meals of the user.
Now at the moment, we got no logic for managing favorites,
and therefore for the moment
I'll just show an empty list here like this.
But this will change later.
But now we have the logic for switching
between two different screens,
and now we can use activePage,
which contains one of these two screens as a value for body.
And with that, we got the logic to change the index.
And based on the index, we change the screen
which we output as body here.
Now we can also set this title here dynamically
by adding another variable activePageTitle,
which by default is Categories, for example.
But if the SelectedPageIndex is one,
we can set activePageTitle equal to Favorites
or your favorites,
and it's now this activePageTitle
that will be output here
in this text widget in the app bar as a title.
And now with all these steps taken,

we can go to the main dart file
and make sure that by default
we don't show the CategoriesScreen,
but instead the newly added TabsScreen.
And for that of course,
you can remove the categories screen import,
but make sure that you added the tabs screen import
like this.

If you now save everything
and force a reload,
you should have some tabs at the bottom here,
and by default, the first tab is selected.
Now if I click the second tab,
you see that indeed I switched to the Favorites screen,
but you might also see
that indeed the second tab is now not highlighted.

It's still the first tab which I can tab to go back,
but the highlighting of the tab never changes.
The reason for that is that we actually have
to tell the bottom navigation bar which tab is selected,
and this is done by setting another parameter.
And indeed bottom navigation bar has many parameters
that can be set to configure this navigation bar
exactly as you want it.

But here we have to set the current index
and set this to our selectedPageIndex
because this will simply control which tab
will be highlighted.
And we wanna keep this in sync with our state
with the selectedPageIndex state.
Once you made this change here,
you can go back

and you will see that now the tabs are selected

and highlighted correctly.

And that is in general how you can add tab-based navigation.

But at the moment, we actually have a problem,

at least a potential problem.

We have two app bars here,

also of course on the Favorites screen.

And this of course is not ideal.

So why do we have these two tab bars?

Well, we have them

because we have a scaffold widget here in the tabs screen,

but we then also have a scaffold widget

in the categories screen here,

where we also add a tab bar.

And in the meals screen where we also add a tab bar.

Now for the Categories screen,

the solution is quite straightforward.

We should simply go to `categories.dart`

and remove the scaffold there

because we are setting a title for this screen

inside of the tabs screen.

There, we are managing this active page title,

which is set as a title for the overall screen.

So in `categories.dart`,

we can remove this scaffold here

with the app bar and body

and remove one closing parenthesis

so that we directly return the grid view.

With that, here on the categories screen,

we now only have one app bar and it still works as before.

So that's all good.

For the Favorites screen,

it's more difficult

because here we're using the meals screen

and we can't just remove the scaffold widget here

because the meals screen is also used

if we dive into categories.

Here, we got the Meals screen as well.

And here we need scaffold

because it is a standalone screen

without tabs at the bottom.

Therefore, we should conditionally use this scaffold widget.

And one way of showing it conditionally would be

to actually make this title optional

by adding a question mark after string

so that it can also be null.

Then we can also remove required here

so that the title does not have to be set.

With that down there,

we can actually add some conditional logic

and check if title is null.

So if it's not set,

and if that's the case, we just return our content.

Otherwise we returned a scaffold that uses the title.

And here we can add an exclamation mark

because we know that if we reach this code here

and this code gets executed,

title will not be null

because if it would be null,

we would return here.

And as you learned, once you return,

any other code thereafter won't be executed.

So we either return content, which is simply our list,

or this fallback text here,

or we return the Scaffold with the AppBar,
and now we can go back to the tabs screen.
And here, when creating the meals screen,
we now simply don't set the title,
we just set the meals
and we can set this to const for now,
but we don't set the title there.
And by not setting the title,
we will make sure that this scaffold
inside of Meals screen won't be displayed.
So now if we reload,
you will see that if you switch to Favorites,
there is only one app bar up there.
And on the other hand, if we dive into a category,
we still have the app bar there as well.
So here we still got the scaffold.

16. Passing Functions Through Multiple Layers of Widgets (for State Management)

Now that we got the tabs working,
we can also work on this favorites feature.
And for that it should be possible
to visit this MealDetailsScreen.
And here I want to have a button
that allows me to mark this meal as a favorite
and to allow users to mark a meal as a item,
I'll go to this meal details dart file
where we have the MealDetailsScreen.
And there in the app bar I want to add some actions
or one action to be precise,
I want to add a icon button as an action,
which can be used to mark a meal as a favorite.
Now for that of course,

we'll have to do something once this is pressed,
but then we also have to add a icon here
and the icon should actually change
based on whether this meal already is a favorite or not.

For a start, I'll simply create a icon which it here,
which is set to icons.star.

And we can set this to const,
though we will change this logic later anyways.
Now with that, we'll have this icon here at the top,
which is nice.

And when we tap this button,
this meal here should either be saved as a favorite
or should be removed from the favorites list.

And this of course means that we need
to manage some state somewhere in this application.

We need to manage a list of meals or meal IDs
or anything like that that have been marked as a favorite.

But where should we manage this list?

Well, not inside of this MealDetailsScreen
because if we would manage this state in here,
we would not have access to it from inside other widgets.

And we do need this information
which meals are a favorite or not
in the tabs screen, for example,
because here we are displaying the MealsScreen
and we're passing the meals
that should be rendered on that MealsScreen to it.

And since we're rendering the MealsScreen
as the favorites screen here,
in the end, we want to pass all the favorite meals
to the MealsScreen in this place.

So therefore, we need access to the information

which meals are marked as favorites
in this tabs screen widget.
Of course, the state is manipulated
from inside MealDetails though,
so therefore we'll have to make sure
that we create a function
that adds or removes favorites here in tabs screen state,
but will have to pass this function down to MealDetails
so that when this icon button is pressed there,
we actually change the state in the tabs screen.
So we're lifting the state up in the end.
Something we have done many times before in this course,
and which we will do again here.
So for that here we can add a final list of meals
and it can be final
because we'll never reassign this property,
which I'm about to add here
and this property can be called favorite meals.
And initially it's an empty list
and meal also should be imported
so you should add this import from the models folder.
So this list is empty initially,
but of course now we can add a new function,
which could be called toggleMealFavoriteStatus
because this function should either add a meal to the list
or remove the meal,
it should be one function that does both.
And here I expect to get a meal as a parameter.
And then in this function,
we need to find out whether this meal here
is already part of this list,
in which case it should be removed,

or if it's not part,
in which case it should be added.
Now we can use different approaches
to determine whether this meal is part of this list or not.
And one simple, straightforward way of determining this
is to add a variable `isExisting`
and in this variable I want to store true or false
but of course that should be derived dynamically,
which I can do by taking a look at `favoriteMeals`
and then calling `contains`,
a built-in method that you can call on any list
to check whether favorite meals
contains this meal, which I'm getting here.
If it does, this will return true,
otherwise it will return false.
And now we can check if `isExisting`
is true or as a shortcut,
we can also just write `isExisting`
since this already stores true or false
and if once true or false
so that's all we have to write.
Checking like this would be redundant.
So here I'm checking if `isExisting` is true in the end.
And if it is true, it means we have to remove the meal.
So in this case, I want to access `favoriteMeals`
and call `remove` and remove the meal.
And otherwise I'll access `favoriteMeals`
and call `add` and add that meal to the favorites.
And this is all we have to do here.
This will make sure that a meal is added or removed.
So now that we have this function,
we have to get it to the `MealDetailsScreen`

because that is in the end
where it should be triggered
when this icon button here is pressed.

The problem of course is that the tabs screen
doesn't have direct access to the MealDetailsScreen.
Instead, the tabs screen only has access to the categories
and MealsScreen.

But both these screens ultimately reach
the MealDetailsScreen.

I mean inside of the MealsScreen,
we simply do push the MealDetailsScreen.

So here we could pass a pointer
to this toggleMealFavoriteStatus function
to the MealDetailsScreen.

But in order to be able to pass a pointer to this function
to the MealDetailsScreen from inside the MealsScreen,
we first have to pass this function from the tabs screen
to the MealsScreen
and later as a second step
also through the CategoriesScreen to the MealsScreen
because that's another place where the MealsScreen is used.

And of course, passing a pointer to a function
through multiple layers of widgets like this
is a bit annoying,
but that's why we're going to learn about
an alternative approach in the next course section.

Nonetheless, this is an important pattern
which you will also see and use a lot,
and therefore we are going to learn about it here.

So in MealDetailsScreen in the end,
I expect to have a property
which will be a function actually,

so which will hold a function as a value
that could be called `onToggleFavorite`.
And this here should be a function
that in the end then gets a meal as an input,
the meal that should be added or removed as a favorite.
So therefore, here in `MealDetailsScreen`,
we have to require this `onToggleFavorite` parameter
to then set the function that we receive on this parameter
as a value for this property.
And then here on the icon button,
inside of this anonymous function,
we can call `onToggleFavorite`
and pass our meal as a value to it.
So this meal which we store in this property,
because `onToggleFavorite` will be a function
that wants a meal
hence we pass this meal as an argument.
Now inside of the `MealsScreen`, we now have to make sure
that we pass such a function to `MealDetails`
because `MealDetails` wants this `onToggleFavorite` parameter,
which wants such a function.
And since we use `MealDetailsScreen` in the `MealsScreen`,
it's this place where we have to provide this function.
Of course, we don't have this function here though.
We have it in the tabs screen instead.
And therefore in the end, in this `MealsScreen`,
we also just have to register such a extra property
and require this extra parameter `onToggleFavorite`,
just so that we can forward it to `MealDetailsScreen`.
Because there indeed we can now add `onToggleFavorite`
and set it to this `onToggleFavorite` property
that we have here.

So now with that, we have to make sure that we pass this function, which we now want to get here in MealsScreen on this parameter to the MealsScreen. So from inside the tabs screen, where we create the MealsScreen, we now have to set onToggleFavorite and here we finally can and now set it to this toggleMealFavoriteStatus function that we create here.

So with that, we're in the end creating a chain along which we pass a pointer to this function through multiple layers of widgets.

Now we have to remove const here, but with that we will pass this function to the MealsScreen and through the MealsScreen in the end to the MealDetailsScreen.

And there in the MealDetailsScreen, we then call it when this icon button is pressed, that's what we're doing here.

Now of course, we're also using this MealsScreen in the CategoriesScreen, so therefore there we also have to accept such a function property.

So we have to add it as a property and accept it as a parameter, I mean, and this must be a required parameter.

Here, I also must import the meal model, so this import here is required to make this meal type available in this file.

And now since we get this function here and store it in this property,

in the CategoriesScreen,
we can set onToggleFavorite on the MealsScreen
and pass this stored value in this property
that's also named onToggleFavorite
to onToggleFavorite on the MealsScreen.
And I know this can be confusing,
we're passing a function through multiple widgets here,
but right now it's the only way of doing this,
it's the only way of getting a function
that's defined in this tabs screen
to the MealDetailsScreen,
as mentioned, we'll learn about a different leaner approach
in the next course section,
but with that we're almost done,
now, as a last step,
we just have to go back to the tabs screen
and there set a value for the onToggleFavorite parameter,
which we just added there,
this parameter, which wants this function.
And finally, we therefore now can also use
our toggleMealFavoriteStatus function here as a value
for this parameter for the CategoriesScreen.
And we have to remove const here.
And with that all, we now have the logic in place
to add or remove meals as favorites
and to trigger that logic
from inside the MealDetailsScreen.
Now, of course, we have to make sure
that we also output any favorites we might have set
here on the favorite screen
and that's what we'll do as our next step.

17. Managing App-wide State & Data

So now that we juggled our function around and passed it through almost all widgets that we have here, we also have to pass the favoriteMeals around. Though here, we don't have to pass that through all our widgets. Instead, we just have to pass our favoriteMeals here to our MealsScreen if it's being displayed as the favorites screen, as it is the case here in our tabs screen if the second tab was selected. Here, instead of passing an empty list, we wanna pass our favoriteMeals. If we do that and reload everything, if we now visit a meal and mark it as a favorite, you will see that now in the favorites screen, this meal shows up. If I then click this button again, it should be removed, but it actually is still there until I navigate away and go back here. So what's the problem here? Well, the problem simply is that in toggleMealFavoriteStatus, I am updating my favoriteMeals, but I'm not calling setState. And of course we have to call setState to make sure that our update here also triggers the build method again,

which then in turn will make sure

that the UI is updated.

So we have to wrap our `favoriteMeals.remove`

or `favoriteMeals.add` calls

into `setState`

instead of `toggleMealFavoriteStatus`.

And if we do that,

you will see that if now I add an item as a favorite

and I go to favorites

and unfavorite it by clicking this icon again,

it indeed is removed instantly.

So that was the only problem here.

But there is one other problem

or potential improvement

that we should make.

And that's this button here.

If I tap it, it never changes,

so I don't really know

whether I added a item as a favorite or not.

It would be nice to get some feedback.

Now, we will make sure that the icon changes

in the next course section,

because right now it would be unnecessarily complex

and it will be much easier

in the next section

once we learn about a more powerful way

of managing cross-widget state,

like this `favoriteMeals` list state,

which does affect multiple widgets.

So therefore, for the moment,

instead of replacing this icon here,

I'll do something different.

In the tabs screen where I have my favoriteMeals
and where I know whether a meal
was just added as a favorite
or removed as a favorite,
I wanna show a little info message
once we did update a meal.
So therefore here I'll add a new method,
showInfoMessage,
where I expect to get the message
of type String as an argument.
And in here I want to show a snack bar,
and you learned about snack bars earlier in the course,
in the expense tracker section.
We can show such a snack bar
through the ScaffoldMessenger,
which needs the context though.
But since we're in a state object here,
we have this globally available context,
so that's quite convenient.
And then here, I first of all wanna clear
all existing snack bars
if we're already showing some.
And then thereafter,
we can show a new snack bar
by calling showSnackBar, like this.
And to show snack bar,
we pass a snack bar
with any content of our choice
where content is a widget.
And here for me, it will just be a text widget
where the text is the message I'm getting here.
Now, we could also set a duration then,

but the default duration
is actually four seconds,
which works for me here.

So I'm just setting the content.

And now we can call `showInfoMessage` from inside here.

So if we just removed a meal
from the `favoriteMeals` list here,
I'll call `showInfoMessage`
with a text of, "Meal is no longer a favorite."

And alternatively, if we added a meal to the `favoriteMeals`,
I'll send a message of,
"Marked as a favorite," something like this.

Now, with that, we get at least a little bit of feedback
if we select an item
and mark it as a favorite
because now we get this snack bar at the bottom,
and we also get that
if we mark this item as no favorite.

So that's quite good, quite convenient.

We get some feedback here, which is good.

Again, we'll work on the icon
in the next core section.

But with that, this favorite feature is working.

We can add and remove favorites as expected.

And now, with that, we can move on
to the last part of this section
and make sure that we add a side drawer
that allows us to switch to a filters page
where we can set some filters
that will influence which meals do show up
if we do select a category here.

18. Adding a Side Drawer

So, now that we have
our tab based navigation working
and we can mark items as favorites
and see them under the Favorites tab,
it's time to add this side drawer,
which I mentioned earlier.

A side drawer which we can open up to go
to a filters screen on which we can set various filters,
which will then impact which meals we see
when we select a category.

And just like tabs, which we added with help
of `bottomNavigationBar` on `Scaffold`,
such a side drawer is an absolutely common
and standard feature,
and therefore very easy to add with Flutter.

Indeed, it's again the `Scaffold` widget
that allows us to add a drawer to this screen
because that's also important to understand.
Side drawers are added on a per screen basis.

If you want to have a button
that opens the side drawer on a specific screen
in the app bar,
you have to add it to the `Scaffold` of that screen.

So, here, if I want to have a side drawer on both tabs,
I have to add the side drawer to the tabs screen scaffold,
and that's exactly where I'm at in my code.

So, there on `Scaffold`, we can then add the drawer parameter.

And as you can already see,
you also got a couple of other drawer related parameters
that allow you to basically configure
the behavior of the drawer

and listen to certain events.

But here, I'll simply add the drawer parameter

and this wants a widget in the end,

a widget that will be presented as the drawer.

Now, for that Flutter has the built-in Drawer widget,

which is simply a widget that's optimized

for being shown as a side drawer.

But since I'm about to add a lot of content to that drawer,

I'll actually create a brand new widget,

my own custom widget, in the widgets folder therefore,

because this will now not act as a screen,

but as the side drawer instead.

And I'll name it `main_drawer.dart`.

Of course, as always, all these names are up to you.

But in this file, I'll then add a new class, `MainDrawer`.

And this class will extend `StatelessWidget`,

because even though this drawer will then provide buttons

that take us to different screens,

we'll again perform this navigation

with help of the Navigator object

and not by setting some state,

hence the Drawer widget can be stateless.

Now, in this drawer class,

we therefore as always add the build method

and also a `const MainDrawer`,

where at the moment I only accept and forward the key

as we always do.

And now here in the build method,

I will return that built-in Drawer widget.

Now this Drawer widget gives you

a bunch of configurational options, like for example,

setting the background color of the drawer

or the color of the background shadow
that will be displayed below the drawer
or with elevation, you can basically define
how much drop shadow you want to have behind the drawer.

But here I'll simply set the child which defines the content
inside of the drawer.

And as I mentioned, the Drawer, which it is already a widget
that's optimized to be presented as a drawer.

So, it will for example, automatically make sure
that it takes the entire screen height and so on.

So, here, as a child, I'll then add a column
because I wanna have multiple items
above each other in that drawer.

And for that of course we set our children here
in this column as we always do.

And then the different children that I do wanna set
are basically a header at the top of the drawer.

And then below that, a couple of clickable or tappable items
that take us to different screens.

Now for the header we can use
the built-in DrawerHeader widget,
which is another widget optimized for its role,
in this case, optimized for being used inside of a drawer
to show them header,
as you might guess, at the top of the drawer.

Now, DrawerHeader can be configured in different ways
and accepts different parameters.

And I will for example also soon add a decoration to it
to style it basically.

And also add some padding, which is supported here
with the padding parameter.

And I'll start with that

and give it a padding of 20 in all directions like this.

And then in addition to this, I'll also re-add decoration and set this to a BoxDecoration object as we did many times before for other widgets as well in this course.

And I'm doing this so that I can add a gradient here.

And I guess that was the most common scenario where I used BoxDecoration thus far in the course for adding gradients to widgets.

And I'm doing the same here.

Now, the gradient is created as we always do it with the LinearGradient class here, provided by Flutter.

And then we can set our colors and then also begin and end.

Now, for the colors, I'll use some colors from my theme with Theme.of(context) using this context object, which we get automatically in the build method.

And then here I'll reach out to my colorScheme and use the primaryContainer as a starting color.

And for the end color, I'll also use colorScheme.primaryContainer.

So I'll copy that, but then I'll add some opacity to it to have a slight gradient between the same colors, but where one color has some transparency.

And then I'll begin with that gradient in the top left corner with Alignment.topLeft, and I'll end in the bottom right corner like this.

And with that I got some styling added to the DrawerHeader.

Of course, it also needs a child though, which simply defines the main content inside of this header.

And for that here, I'll add a row as a child because I wanna have multiple items next to each other

inside of that DrawerHeader.

And the items I do want to have next to each other here are an Icon and then a SizedBox to add some spacing and then some Text.

Now, for the spacing, I'll use a width of 18 pixels.

For the Icon, I chose the fastfood, which is a built-in icon of course that looks like this.

And I'll actually change the size of this icon to make it quite big by assigning 48 pixels as a size.

And I'll also change the color of this icon, and again, use my theme for that to reach out to Theme.of(context).colorScheme, and then there choose the primary color.

Now, for the Text, I'll simply give it a text of Cooking Up like this, and also set a style here where I will again use the theme, and then of course the textTheme property on the theme, and therefore the textTheme object to access the titleLarge styling.

But I want to adjust it a little bit, hence I'll add an exclamation mark to tell Dart that I know that titleLarge will be defined in this theme to then call copyWith and override some settings, specifically the color where I will again use the theme, and of course as always, the colorScheme thereafter, and then the primary color.

So, that's the content of this row, which in turn is the content of the header.

And with that, before we add anything else, I wanna see whether this drawer is displayed correctly and whether this header looks good by going back to tabs.dart.

And here for this drawer parameter on Scaffold,
I will add my MainDrawer widget here.
And we can add const in front of this.
Of course, we must make sure
that main_drawer.dart is imported in this tabs.dart file.
But with this import added
and the drawer is set on the Scaffold
as we're doing it here, we can save everything.
Make sure that the app is up and running.
And if you go back to your app,
you should now see this menu icon here
in the top left corner on Android at least.
And if you tap it, you open up this drawer
where you see this header that we added,
which looks the way I want it to look.
And if we tap on the background
or we slide this in, we can close this drawer.
So, that works as it should, and that's awesome.
And we can also access this from the Favorites page
because that's still part of the tabs page,
though we cannot access it if we dive into a category.
For example, there we have this back button as we want it.
So this all works as intended.
And therefore, now the next step is to add some items
to that drawer, some tappable items
that take us to different screens.
So in order to go to different screens,
I'll go back to the MainDrawer here
and they are below this DrawerHeader in the children list
of this Column here.
So, here after DrawerHeader,
right before I close this list of children for the Column,

I will add more content.

Specifically, I'll add the links

to the different screens I wanna support in this app.

And for that I'll use another built-in widget,

which is not exclusive to the drawer,

but which you can use in any list you might be outputting.

And that would be the ListTile widget.

I haven't really used this widget a lot thus far,

but it can be a great widget for outputting list content

since it's simply optimized for bundling

different pieces of information together in a single row.

Here, I'll use Flutter's ListTile widget

to output my drawer links.

For that, the ListTile widget accepts multiple parameters

that offer you rich configuration options.

And you can, for example, set a title here

to add the main content of this ListTile in the end.

Any widget is accepted here.

And here, I'll simply go for the Text widget

to then output my different, well, drawer link texts.

For example, meals for the first list item,

so that this would be the list item that takes us back

to this tabs screen, for example.

So, that's the title I'm setting here.

And I'll add a style here again by reaching out to my theme.

So, from that theme, I'm going to grab some styling.

Here, I'll not use titleLarge,

but instead titleSmall and see how that looks like.

And then I'll set my color here,

my text color, to onBackground.

And I'll also override the font size and set it to 24.

So, with that, I set this text on this first ListTile.

That's not all I wanna set though.

I also want to add an icon

in front of this text essentially.

And for that we could set title to a row

where we then manually add an icon next to a text.

But the ListTile actually also gives us a leading parameter,

which allows us to set a widget

that will be presented at the beginning of this row

of this list item.

So that this title comes after the leading widget.

As I mentioned, ListTile simply is a built-in widget

that's optimized for outputting items in a list.

So inside of a row that's part of a list.

And leading allows us to add some widget

that will be displayed at the start of that row.

And here the leading widget I wanna output is an icon.

And for this Meals item here, for this Meals link,

I'll use the built-in restaurant icon.

I'll then also set the size of this icon to 26

and change the color of the icon

and set that to Theme.of(context).colorScheme.onBackground.

So the same color I used for my text down there.

Now with that, we would be displaying one link

in our drawer, but of course it wouldn't be useful

if it wouldn't be tappable.

And therefore, of course, besides adding leading and title,

I'll add one last very important parameter to ListTile,

and that's the onTap parameter,

which wants a function as a value.

And later, we'll make sure

that when this function gets triggered,

we navigate to a different screen.

So, that's the first step.

The next step is to copy this ListTile
and add a second one.

But in this one here we'll go to the filters screen,
which we have yet to add.

And I'll use a different icon here,
the settings icon to be precise.

But besides that, it's the same configuration as before.

And therefore of course we could outsource this
into a separate reusable widget.

But here, I'll just go with copy and pasting.

And with that I added the two items I want to have
in that drawer.

And if you now save that,
you should be able to open that drawer
and see these two items and they are tappable.
But of course at the moment, nothing happens when I tap them
because I haven't added any logic
to those tap functions here.
But that is what we'll do next.

19. Closing the Drawer Manually

So let's now make sure
that if we tap those drawer items,
we actually navigate to a different screen.
Now for that, we could go to these anonymous functions
that are set as values on tap
and indeed use `Navigator.of(context)`
to then push a different screen on top
of the screen stack right from inside this drawer.
And this wouldn't necessarily be wrong,
but it would cause one problem here.

In this app here, we will, in the end,
need to manage the active filters
that have been chosen on that
to be added filters screen.

We will have to manage those
inside of the tabs.dart file
because it's inside of this tabs screen widget here
where we, in the end, load the CategoriesScreen.

And the CategoriesScreen then, in turn,
will load the MealsScreen, if you remember.

Inside of categories.dart,
we push the MealsScreen.

And the MealsScreen will need to know
which meals should be displayed.

And therefore, of course, in the future,
we'll have to make sure that we pass this information
about which filters have been selected
to this MealsScreen
and therefore, in the end,
also to this CategoriesScreen.

And therefore, we will need that information
inside of this tabs screen.

Because that is our main data management widget in the end,
because it's this tabs screen widget
that loads all the other meal-related screens
that needs that meals and filters information.

And for that reason, I'll not do the navigation
from inside my MainDrawer widget,
but instead in this MainDrawer widget,
I'll add a new property function,
which will need some string identifier as a parameter.

And you'll see what this parameter

is all about in just a second.

And I'll name it `onSelectScreen` or anything like this.

And of course, then I'll make sure

that this parameter of the same name

is required and bound to this property.

Now the idea is that this function,

which we store in this property,

can be called from inside `onTap` here

for these two list items.

And here for this first list item

where we want to go to the `MealsScreen`,

I'll pass an identifier of meals.

And on this second `ListTile`

where we want to go to the filters screen,

I'll call `onSelectScreen`

and pass filters as a value.

That's the idea behind this string identifier.

And the reason for adding such a string identifier here

is to make sure that in `tabs.dart`

we can define this function,

which we expect as a value here in `MainDrawer`,

and which we then bind to this `onSelectScreen`.

And in `tabs.dart`, we then get this identifier

to decide which screen should actually be loaded.

And then we can manage and pass the data

that might be required by that screen accordingly.

So therefore here in `tabs.dart`,

I'll add a new function,

which I'll name `_setScreen`.

And here I'll expect to get this string identifier.

And it's this `_setScreen` function

that will be passed as a value to `MainDrawer`

under this onSelectScreen parameter, which we added.

And now we have to remove const here.

So with that, the MainDrawer can, in the end,
communicate with the tabs screen.

And it will be this _setScreen function,
or method to be precise, that will be executed
if one of these ListTiles will be pressed.

And we will simply get a different identifier value
inside of this _setScreen function,
depending on which drawer item was pressed.

And here we can then check, for example,
if identifier == filters
to then load the filters screen,
which we have yet to add.

And else, we know it must be meals.

And in this case, we can load the tabs screen.

Now for this else case here,
so if you want to go to the tabs screen in the end,
it's, of course, important to keep in mind
that we already are on this tabs screen here.

So if we open the drawer from this tabs screen
and we then selected meals in that drawer,
we effectively just want to close the drawer.

Because we already are on that meals screen
or in that meals area of the app already.

And therefore, if we just wanna close the drawer,
we can do this through the Navigator object
and close it just as we close modals or dialogues
before in this course.

We can simply use pop like this.

The context is available here
because we're in a state class,

in a class based on state here.

So that's how we can close the drawer
if we are on the meals screen,
on the tabs screen in the end,
and we want to go back to the meals area.

And therefore if you save this and force a reload,
you will see that if I open this
and I click Meals,
I simply close the drawer.

So this works as it should.

So now let's make sure
that we can also go to the filters screen.

And for that, we, first of all,
have to add such a filters screen.

20. Adding a Filter Item

Now we need a new screen,
the filters screen,
and therefore in this screens folder,
I'll add filters.dart which as you might guess
will hold my filters screen widget,
and this actually must be
a state full widget here
because we'll need to manage
some internal user input state
in this widget.

Since we will present multiple switches
to the user, that can be used by the user
to set various filters.

So here we have the stateful widget,
and of course, as always,
we have the create state method here,

and we also have filters screen
as a constructor here
where I forward my key like this.
And then I'll add the underscore filters,
screen state class
where I extend state for filters
screen like that,
like we did before.
And also use that here
as a value for this generic type.
Then here we add the build method.
Now here in create state,
we then return a new instance
of our filters screen state class,
and in the build method down here,
we then return a scaffold.
Because this is a brand new screen,
and the offer,
it gets its own scaffold so that we for example
can set an app bar for that screen,
but also so that we soon can add the drawer
because this screen
will also need access to the drawer
so that we can leave this screen.
But I'll get back to this in a second.
So here we have the app bar,
and I'll add a title here on this app bar,
which I'll set to a constant text widget
where I simply say, your filters,
this sounds like a fitting title.
And then of course, we also need a body here.
So the main content of the filters screen.

And here I'll go for a column
because I wanna display multiple filter switches
in the end.

Switches that will be shown below each other
that can be used
to turn these different filters on or off.

So therefore here I add children.

And now every switch
will be a switch list tile,
which is a convenience widget
provided by Flutter that's optimized
for being presented in a list
because this will then display
one single row in that list,
but in a list full of switches
where you can set a label
for every switch, but you'll then also have
that switch widget that can be tapped by the user
to turn something on or off.

So that's a very convenient widget for rendering
for example, setting lists
as we're doing it here in the end.

And switch list
tile wants a value, which is a boolean,
so which is simply true or false,
controlling whether the switch is currently
turned on or off.

And then an onChanged parameter
that in the end wants a function
that receives a boolean,
which will be triggered whenever
such a switch is pressed.

Now I will add both in a second,
but before I do that,
I'll add some other parameters,
which I also wanna set on a switch list tile.
For example, the title.
The title parameter can be used to add a label
that will be shown next to the switch.
And here I wanna display some text
with help of the text widget.
And the text I do wanna display here
for my first switch is gluten-free,
because this first switch
will be about setting this filter.
I then also wanna apply some styling
to this texture.
And for this I'll use my theme,
the text theme and the title large.
And again, copy this
with some extra settings, which I'll change.
And the setting I will change here
will be the color,
which I also take from my theme,
but where I then want
to use the on background color
from my color scheme like this.
Now in addition to setting this title,
I'll also set a subtitle,
which as the name kind of suggests,
is a parameter.
We can set on switch list tile
to render some typically smaller text
below the main title,

the main label of the switch, so to say.

And in this case I'll use subtitle

to render some extra explanation.

And for my first switch here

where I toggle the gluten-free filter,

I'll say Only include gluten-free meals.

This is simply some extra explanation

of which I want to add here.

And I will also adjust the style

of this text here.

And hence I'll copy the style from my title.

But here I don't want to use title large

as a starting point,

but instead label medium,

which simply displays smaller text.

But I'll also set the color

to on background here.

Thereafter, I'll also set

the active color parameter,

which allows us to control

which color will be used

for rendering this switch if it's activated,

if it is turned on, so to say.

And here I'll also use a color from my theme,

and I'll use the tertiary color here.

Last but not least,

I'll set the content padding parameter

to control how much padding

will be used inside of this list tile in the end.

And here, I'll set this to edge insets only

to only target specific directions.

And I'll set a padding to the left of 34,

and to the right of 22.

And of course you can tweak,

and fine tune these values as needed.

So that's the base configuration

for one row in my list of settings,

which I will in the end have.

And now of course we have to come back

to the value and unchanged.

Now the value will be some state

that I manage here

because of course the value should be changeable.

And if it is changed,

it should be reflected

back on the user interface.

So here I'll add a variable

to the filters screen state,

which I'll call `_glutenFreeFilterSet`.

And initially this could be false,

so that initially the filter

is not set though soon,

we're going to load the data

that has been set by the user in the past.

But at the moment,

I'll always initialize this to false,

and then use this as a value down there.

And then for unchanged,

I'll add an anonymous function.

We could also add a separate method

to this state class of course,

but here I'll go for an anonymous function

where I get the new value

or simply the answer to the question,

whether this is checked or not.

This parameter name is up to you

because it's your function,

but it will always be a boolean

because this function will be called

by Flutter for you,

and Flutter will pass a boolean value

to that function if you want it or not.

But now here I can use this

is checked information

to update `glutenFreeFilterSet`.

And for that,

of course we have to call `setState`

because the UI should reflect that change.

And then I'll set,

`_glutenFreeFilterSet = isChecked`

so to this parameter value here.

And with that, we should have a first list item.

21. Replacing Screens (Instead of Pushing)

Now before we add all the other list items,

I'll save this and make sure that we can actually see this.

And of course, to see this we have to make sure

that this `FiltersScreen` actually gets loaded.

For that, we can go to `tabs.dart`, and here in `setScreen`,

if the identifier is set to `filters`,

I do want to go to this `FiltersScreen`.

Now of course, we can do this as we did it in the past

by using `Navigator.of(context)`

and then push a new `MaterialPageRoute`,

which has this builder function where we get the context

and then return the screen widget that should be loaded.

In this case, the `FiltersScreen` widget,
which at the moment can be set to a const value.
Now of course, as you know,
in order to use `FiltersScreen` here,
you also must make sure that you are importing `filters.dart`
into this `tabs.dart` file.

With that done, you can load it like this.

And if we would add this code and save everything
and restart, you will see that now if you tap `Filters`,
indeed you're taken to the `Filters` screen.

Now here on this `Filters` screen,
we then have this list item.

And here we can toggle this setting as expected.

And we do have this back button,
and if we press this, we're taken back.

But of course here, the drawer is still open,
which is rather strange.

Now of course, it is still open because we never closed it.

We just navigated to a different screen.

Now what we can do, is we can go to the `Tabs` screen,
and before we navigate to the `FiltersScreen`,
we can call this code here to close our drawer.

So, to pop the drawer, and before we then navigate away.

And now, since we actually have the same kind of code
in both branches, we can simplify this code
and always execute it outside of the `if` statement
right at the start of `setScreen`,

since we always want to close that drawer.

That never changes. It does not depend on this condition.

And we can therefore also get rid of this `else` branch here,
so that we always close the drawer.

But if the identifier is `filters`,

we then also navigate to a different screen.

With that logic added, if you open the drawer and go to Filters, you reach that screen.

But if you go back, you see the drawer is not open.

And that is a pretty nice behavior I'd say.

Now you could also want an alternative behavior, where you also have a drawer inside of your FiltersScreen.

For that in filters.dart, we could go to scaffold, and of course also add our MainDrawer here.

We then also of course, need to add a setScreen method to this class here,

and pass it to MainDrawer,

so that we can react to an item being tapped.

So, we have to set this onSelectScreen method here, where we accept this identifier.

And we can do this with an anonymous function as I'm doing it here, or add a method to this class.

Here, I'll just go with an anonymous function.

And then here, we could basically do the same as before.

Always call Navigator.of(context).pop.

But if the identifier is equal to meals, in this case, since we're on the FiltersScreen,

we wanna go back to the TabsScreen.

And this could be achieved by using Navigator.of(context).push MaterialPageRoute.

And here we got the context, and I want to go to the TabsScreen like this.

Now of course, as you know, for this to work, you also must make sure that you added the appropriate import for TabsScreen, so that you are importing tabs.dart.

And if you do that, you will see that now,
if you go to this Filter screen, the back button is gone
and it was indeed replaced by the drawer button here.
And here you can then go to Meals, for example.
So, that's now simply a different way of navigating around.
Now what you'll notice though,
is that if you use the physical back button on your device,
which on the simulator here on the Android Emulator,
can be emulated here with help of that toolbar on the right,
you indeed can go back through all these navigation actions
that you performed.
Because of course, you did still push these screens
on this stack of screens.
You just don't have this back button here in your app,
because you are setting a drawer.
And hence this drawer, this drawer button,
is used instead of the back button.
But technically, there still is this stack
of screens going on.
And that might be the behavior you want,
being able to go back with help of the back button.
But you also might want to replace the current screen
with the new screen if you're using the drawer.
So that if you are on the TabsScreen
and you go to the FiltersScreen, you actually don't push
this screen onto the stack of screens,
but instead you replace the currently active screen
with the FiltersScreen,
so that the back button would not work
and would not take you back.
And since this is also a pattern you should know,
I'm also going to show you how this could be implemented.

Because it's really straightforward to do.

You just replace push, for example,
here in the TabsScreen file, with pushReplacement.

And the rest stays as it is.

But this will now make sure that FiltersScreen is not pushed
as a new screen onto the stack of screens,
but that instead the currently active screen,
in this case TabsScreen will be replaced
with the FiltersScreen.

And therefore, the back button wouldn't work
because there is nowhere to go back to.

And you can do the same in the FiltersScreen.

There, when you navigate to the TabsScreen,
you could use pushReplacement instead of push.

If you make those changes and you restart your app,
you will see that the drawer still works as before.

But if I go to the FiltersScreen
and I then press the back button, I actually close the app
because there was no screen to go back to.

And then the default behavior indeed is to close the app.

So that's simply an alternative navigation pattern,
which you should also be aware of,
which is why I'm showing it to you.

Now here, I actually don't want to use that pattern though.

Indeed, I don't even want to have this drawer button
on my FiltersScreen, though you could absolutely use it.

But here I don't want to have it.

I'll comment it out so that you still see
how that code could look like.

But I'll comment it out
because I'm happy with having a back button
on the Filters screen to go back to the Tabs screen.

But of course, this also means not only that these imports aren't needed anymore, but that we also have to make sure that the back button on the FiltersScreen can actually work, by ensuring that we don't use pushReplacement here on the Tabs screen, but just push, so that we do push the FiltersScreen on top of this stack of screens, so that there, we will get this back button, which then will actually work. And with all of that, if we save this, we should have the same behavior as before, where we can go to Filters, change something, go back, and this all works. Of course, right now our filter state isn't saved, and we only have one item here on the FiltersScreen. But these are the things we're now going to work on next.

22. Adding More Filter Options

So on this filters screen, we have one switch list tile for the gluten free filter. But of course that's not enough. I have three other settings, which I also wanna be able to set from there. And therefore I will copy this. And again, we could outsource it into a separate widget instead. But to save some time, I'll just copy it here. And my second switch list tile here, will be all about controlling the lactose-free filter. So I'll adjust the title here accordingly. Of course, the styling is the same,

so it's really only the text that changes,
also for the subtitle,
and we'll need to manage some new state,
hence I'll add a new variable here,
the lactose-free filter set variable.
And we can also already add variables
for the other filters that can be set.
For example, the vegetarian filter and the vegan filter.
And then we can go down there again,
to the switch list tiles.
And for the lactose-free one, I'll change the value
to use this new underscore lactose-free filter set variable,
and also update that variable if the list tile
is tapped or changed to be precise.
And then I'll copy this switch list tile again,
also at a comma here, but copy it again.
And now add the vegetarian filter here.
Only include vegetarian meals.
And then last but not least,
add the vegan filter where I only add vegan meals.
And I also need to update those values
and state updating functions.
So for the vegetarian list item here,
I refer to the vegetarian filter set
and update that variable here.
And for vegan, it's the vegan filter set,
which is also updated here.
Now with that,
we should actually see all these options here.
If I go to filters, yes,
and all these options work independent from each other,
which is of course quite nice.

Now we just need to make sure that these settings here are also saved when we navigate back, and then also do affect what we see here when we do choose a category. For example, I don't wanna see a hamburger if I actually selected vegetarian or vegan. So that's what we'll work on next.

23. Replacing WillPopScope with PopScope

In recent versions of Flutter, the WillPopScope widget we're using in the next lecture has been marked as deprecated.

Therefore, in the next lecture, instead of using WillPopScope you should use PopScope.

The code we write in the next lecture should look like this:

```
PopScope( canPop: false, onPopInvokedWithResult: (bool didPop, dynamic result) { if(didPop) return; Navigator.of(context).pop({ Filter.glutenFree: _glutenFreeFilterSet, Filter.lactoseFree: _lactoseFreeFilterSet, Filter.vegetarian: _vegetarianFilterSet, Filter.vegan: _veganFilterSet, }); }, child: Column(...) // same code as shown in the next lecture ), NOT like this:
```

```
WillPopScope( onWillPop: () async { Navigator.of(context).pop({ Filter.glutenFree: _glutenFreeFilterSet, Filter.lactoseFree: _lactoseFreeFilterSet, Filter.vegetarian: _vegetarianFilterSet, Filter.vegan: _veganFilterSet, }); return false; }, child: Column(...) // same code as before ),
```

24. Returning Data When Leaving a Screen

So we got this filters screen

where we can set our filters.

Now we wanna make sure that if this back button is pressed, or also of course if the physical back button on the device might be pressed, we pass those selected filters back to the tabs screen where we then use them to, in the end, narrow down which meals will be shown if we, for example, select the hamburgers category.

Now, therefore, we must find a way of passing those variable values that are managed here in FiltersStart in the FiltersScreenState back to the tabs screen.

And for that of course, we need to find a way of knowing when the user tapped the back button, the physical one, or the one in the top left corner. And this is another common requirement, and therefore Flutter helps you with that. It gives you a special widget, which you can use here in your scaffold to wrap it around your body. And therefore that's what I'll do. I'll wrap a widget around my column here. And the widget I will wrap around it is the WillPopScope widget. WillPopScope is a utility widget that, as you can see, essentially wraps the entire content of this screen here. And WillPopScope allows you and actually wants you to set a onWillPop parameter. And onWillPop actually wants a function that returns a future. And you might recall I mentioned futures earlier. Futures are these objects which eventually resolve to other values. Values which are not there yet. So a function that returns such a future. And that function will be invoked by Flutter whenever the user tries to leave this screen. So here we can pass a function and this function receives no arguments, but in this function here, we can now manually call Navigator.of(context).pop. And the advantage of doing so is that pop actually allows us to pass some data along with it. Some data, which will then be available in the place

where we navigated to this screen,

so to the filters screen.

Well, in that place is our tabs screen.

There, it's in this `setScreen` method

where we in the end navigate

to the filters screen.

So therefore it will be here where we will get access

to the data returned with help of `pop` from

inside this filters screen.

Now here you can return any kind of data

and what I will return here will be a map where I want

to have different keys for my different filters,

and then true or false, depending on whether

that filter has been set or not.

Now, for that to standardize those keys,

I'll actually add a enum in this `FiltersStart` file

at the very top of the file, which I'll name `Filter`.

And here I'll add some unique identifiers

for all these filters that we can set.

So here I'll go for `glutenFree`, `lactoseFree`,

`vegetarian`,

and `vegan`,

so that we have these clearly defined keys.

And these should be my keys in this map,

which I wanna return.

So my map here,

which I'm returning in `pop`,

will have those different enum values as keys.

So `Filter.glutenFree` as a key, for example,

and then as a value separated by a colon

as you learned, the currently stored value for that filter,

which for this `glutenFreeFilter` would be stored in this

_glutenFreeFilterSet variable.

So that's the value I wanna set here.

And I'll now repeat this for all my other filters here.

So for lactoseFree, for vegetarian,

and for vegan like this.

And therefore this map here will be in the end

what will be returned by this Filters screen

when it's popped.

So when the back button is pressed.

And we will then be able to access this data,

this map here from inside the tabs screen

because data is where we navigated to this filters screen.

Now we're not entirely done yet though.

Instead here in this WillPopScope widget,

in this function here, we have to return true or false

to in the end confirm whether we want

to navigate back or not.

And since we are navigating back manually here,

we should indeed return false

so that we're not popping twice.

If you would not be popping manually here,

but doing anything else, like saving data to some database,

it would make sense to return true

instead, if you wanted to allow the user

to leave the screen, here, we'll return false.

Since we already have our leave screen logic here

and we don't wanna pop twice,

because that would essentially mean

that we would probably close our app.

So here we return false,

but here we pop manually, essentially leaving the screen

with this data being passed back.

And now we just have to keep in mind
that `onWillPop` actually wants a function
that returns a future,
which eventually yields a Boolean,
instead of getting a Boolean right away.
And therefore, to make sure that this is the case,
we should add the `async` keyword here
between this empty parameter list
and the opening curly brace.
Because by adding `async` here,
you can not only use the `await` keyword as you learned
before in this course,
but instead you will also automatically wrap the value
returned here into such a future.
So that's a convenient way of creating a future
that will be returned by your function.
So by adding `async` here, this function will return a future
that will then eventually resolve to this false value.
And with that, we have control over this going back process
and we pop manually with our data.
Now we just need to read that data in that place
where we navigate it to the `FiltersScreen`.

25. Reading & Using Returned Data

Now to read that data
that might be passed back by pop,
we can go to this place where we called `push`.
And here it's interesting to see that this `push` method,
which we're calling actually returns a value.
It returns a future.
Thus far in this course we haven't used dart value,
but that's now going to change,

because we'll get a future here,
which, will in the end,
resolve to the data returned by the screen
where we navigate to.
So to this map, in this case.
We'll get a future that contains this map,
and it will be a future instead
of getting this map right away.
Because of course when we push this screen
onto this stack of screen,
we don't immediately get back that data.
Instead, the user may interact with that screen
and may press the back button after 10 seconds,
10 minutes or 10 hours.
And that's why it's a future.
We don't have the return value available immediately,
but at some point in the future,
once the user did navigate back.
So that's why we get a future here.
And to use this future in a convenient way,
we can add `async` here between parameter list
and curly brace,
so that here we can use `await`,
so that in the end then,
we got the result stored in this result variable.
But this will only be set once the user did navigate back.
So once this future did resolve,
we wait for that to happen here
with help of the `await` keyword, so to say.
Therefore, in the next line thereafter,
result will be available
and will be set to the returned data.

Now, right now the type of that data is dynamic,
because here in this file,
we don't actually know which data will be returned
by this FiltersScreen.

Now since we're the developer of this app,
we actually do know
and we do know that it will be of type map,
where the keys will be of this filter type
and the values will be Boolean,
and therefore we can pass this extra knowledge here to dart
as well by adding angled brackets after push,
because push is a so-called generic method.

You can tell by these angled brackets,
which you see here,
just as List was a generic type.

With List, you could, for example,
define which kind of values would be stored in a list.

Here with the angled brackets
after this generic push method,
you can define which kind of value will be returned by push.

Because you can see here
that the generic type you define after push,
this type T, as it's named here,
will be the type of data
the future will resolve to in the end.

So here we can tell dart that it will be a map.

A map, which itself is a generic type
where our keys will be of type Filter.

And this is the Filter enum defined in filters.dart
at the top of this screen.

It's available in tabs.dart,
because we are importing filters.dart here.

So that's the key type of this map.
The value type will be bool,
because we'll have a bunch of Boolean values in this map.
With that, indeed dart does now know
that this result eventually will be such a map
that has filter keys and Boolean values.
And therefore for the moment,
I'll just print this result
to see whether this works as expected.
Later, we'll of course do more meaningful things with it.
But for the moment I'll print it,
save everything, and reload this app.
And now if we go to Filters and set,
let's say Gluten-free and Vegan,
and I go back,
this works.
But if I now open my debug console,
I also have this log message here,
where I indeed see that I got a map.
You can tell by these curly braces,
with my different filter keys,
and then true for glutenFree,
and true for vegan.
A bit hard to read.
It's cut off, but it's true here,
and false for lactoseFree and vegetarian,
which is exactly what I said.
So passing this data from the FiltersScreen
to the tabs screen works.
Now we just have to make sure that we also use
that data here in the tabs screen.

26. Applying Filters

So now that we're getting our filters here,
we have to manage them in the tabsScreen
and we then have to make sure that we pass the updated list
of availableMeals based on the filters that were set
to the categoriesScreen.

Because in that categoriesScreen,
we then need to pass the filtered meals down
to the meals screen.

And in the future we'll have
to filter not just based on the selected category,
but also on the filters that have been set.

So for that, back here in tabs.dart,
I'll get rid of this line where we printed
our selectedFilters.

And I will instead add a new variable
to this TabsScreenState
where I wanna store the selectedFilters.

This new variable will be of type Map,
and it will be a map where the keys are of this filter type.

And the values are Booleans.

And we can name this variable selectedFilters like this.

And initially this should be a map
where I also set all these filters, so glutenFree and on.

But initially I'll set this to false
and it will then be updated once some filters have been set.

So we have glutenFree, lactoseFree,
which initially is false.

So the filter is not enabled.

That is what false means.

Vegetarian is set to false
and also vegan is set to false.

That's the initial value here.

And this value should now be updated
whenever we receive new information from the filtersScreen.

So therefore here in setScreen where we get this result,
and this result is just such a map of filters
as you learned, we want to set the selectedFilters
equal to the selected result.

Now storing the filters like this is not enough
because we also should make sure
that the build method is executed again
so that the updated filters
or an updated list of availableMeals
is passed to the categoriesScreen.

By the way, not to the meals screen
because I don't want to filter the meals
that are visible on that screen.

Of course you could filter there as well,
but my logic is
that favorites should always be visible
no matter which filters were selected.

That's why I only worry about the categoriesScreen.

Therefore here in setScreen, I'll call setState
and set my filters like this.

And here we now also have the problem
that result could be null.

And null is not an allowed value here for selectedFilters.

And therefore here I'll use a special operator,
the double question mark operator,
which is an operator built into Dart,
which simply checks whether the value in front of it
is null, and if it is null, the fallback value defined
after the double question marks will be used.

So this operator allows you
to set up a conditional fallback value
in case this value should be null.
And the fallback value I wanna use here
if result should be null is essentially this map
of initial filters again,
and therefore to avoid some code duplication,
we can actually also add a new variable,
a new global variable, or even a constant,
which we're never going to change internally,
that could be named `kInitialFilters`
k at the beginning to kind of follow this Flutter convention
of naming global variables with a k at the beginning,
technically it's not required,
but this would be my `InitialFilters`
and that's now what I use here
for initializing `selectedFilters`.
But what we can also use as a fallback value down there
after the double question mark.
And with that we're setting the `selectedFilters`
whenever we come back from the `FiltersScreen`.
And now we can use these `selectedFilters`
to update the meals we're showing once we picked a category
on the `CategoriesScreen`.
So now when it comes to passing the filter data
to the `CategoriesScreen`, we have two main options.
We can either pass a list of `availableMeals`,
which then can be picked up
and used from inside this `CategoriesScreen`.
Or we pass the filters
and add the logic for filtering out certain meals
to this `CategoriesScreen`.

And I'll go for the former option
where I actually create an already filtered list
of availableMeals, which can then filter it a bit more
for a fitting category instead of the CategoriesScreen.
So therefore here I'll add a new final variable,
availableMeals, which in the end are my dummyMeals,
which are coming from the dummy_data.dart file,
which therefore also must be imported
into this tabs.dart file.

And now the availableMeals here
should be the dummyMeals filtered such
that the selectedFilters here are taken into account.
Now for that we can use the where method as you learned,
which exists on all list objects
and which allows us to filter a list.
We pass this function to where, which you saw before,
which is executed on every list item,
so on every meal.

And we receive that value inside of this function.

And then in here we wanna return true
if the meal should be kept and false,
otherwise if it should be dropped.

But of course this should now be derived dynamically
and two derived dynamically.

I'll add a if check in there in this anonymous function
that's passed to where,
where I take a look at my selectedFilters
and I take a look at the glutenFree filter, for example,
by accessing that value by its key.

And that will be true or false,
true if the filter is set, false otherwise.

So with this code here,

I'm essentially checking
whether the glutenFree filter is set
and I combine this with another condition
with this double ampersand operator, which I introduced
before in the course to check that if this filter is set,
I want to take a look if this meal maybe is not gluten free.
Now the meal has a isGlutenFree flag,
a property called isGlutenFree, which is true or false,
and I wanna check if that is not true
by adding an exclamation mark in front of this.
that's how you can check for the opposite
of a certain value in Dart.
So isGlutenFree is true or false?
And I want to check whether it is not true.
And I do this by adding an exclamation mark here.
So if I only want to show meals that are gluten free
and this meal here is not gluten free,
then I don't want to include it.
So then I'll return false.
That's what I'm doing here.
And here I also need to add an exclamation mark
after this code, which does now not check for the opposite.
This only happens if you add it in front of a value.
If you add it after a value,
it's simply a Dart feature which you saw before,
which tells dart that this value in front
of the exclamation mark will never be null.
And I know that this will never be null
because selectedFilters will always be set
to at least this starting map, which has all these keys.
So with that, I'm checking whether this filter is set
and whether our meal maybe does not satisfy this filter,

in which case I return false and exclude this meal.

And now this logic can be repeated for all filters.

So I can also check for the lactoseFree filter

if this is set and active.

And if it is, I want to exclude meals

that are not lactose free.

And then I repeat this again for vegetarian

and I want to exclude meals that are not vegetarian.

And last but not least, of course, also for vegan,

exclude meals that are not vegan.

That is the logic here.

And with that, availableMeals will be a list

that only includes meals that meet our filter conditions

or to be precise, it will be an iterable.

And therefore here I will call to lists to make sure

that I actually return a list and not an iterable.

And I need to add one more return statement here

in this anonymous function that's passed to where

where I return true.

Otherwise I just have a bunch of checks

that potentially return false,

but I never return true for the meals

that I actually want to keep.

And of course, I definitely do want to keep all meals

that don't make it into these if blocks,

so that meet all the filters set

and that's why return true is needed here at the end.

And with that, we're getting this availableMeals list

and it's now this list that should be passed

to the categoriesScreen.

So there we can add a new property,

a list which is full of meals,

and it could be named availableMeals.

Sounds like a fitting name.

We of course also added here

as a required parameter like this,

and now it's the availableMeals instead

of all the dummyMeals that will use as a starting point

for finding meals that fit the category that was selected.

So here we now use availableMeals instead of dummyMeals.

With that added, with this parameter added

to the categoriesScreen, we can go back to the tabsScreen

and there when we create the categoriesScreen,

we can set the availableMeals

to the availableMeals we are deriving here.

So to the value stored in this variable like this.

And now with that, last but not least,

we should be able to filter our meals as expected.

If I now go to filters

and I for example, only show vegetarian meals,

so I exclude all meals that are not vegetarian,

I don't see anything if I go to hamburgers.

And under quick and easy, I also have less meals available.

So this now indeed does work the way it should work.

And if I set more filters like gluten free and vegan

and also lactose free, I of course see even less meals

because now I narrowed this down even more.

And that of course, therefore works as it should.

The only thing that's not working is that if I go back

to the filters screen,

all filters are reset back to nothing.

So therefore thereafter I also see all meals again.

And that's of course not how it should work.

Instead, the filters I set here should be saved

so that they are also shown.

Again, if I go back to the filtersScreen.

Now, to save those filters

and to not reset them, if I go back to the filtersScreen,

we have to make sure that from the tabsScreen

we pass the currently active filters to the filtersScreen

so that they're, instead of initializing all the filters

as false as we're currently doing it,

we're actually initializing those filters to the values

that have been set before.

And to do that, we can go to the filtersScreen here

and add a new property

and parameter to our filterScreen widget.

The property can be named currentFilters

and the type of value stored in this property

should be this map of filters like this.

And of course then I expect to get this as a parameter.

Now that we do get this, we can use the current filters

to initialize our filter state variables here.

Now, since we are technically in a different class here,

we're not in the widget class,

but in the state class,

we can't access currentFilters like this here.

Instead, you learned earlier

that there is this special widget property

which you can use to access the properties

and methods of the widget class, like currentFilters.

The only problem with that is

that the widget property is not available here

where you are initializing your class variables.

Instead, it's only available inside

of the methods of this class.

Therefore, what I'll do as a workaround is I'll initialize all those values as false again.

But I will add this initState method, which you also saw earlier in the course already.

And here after calling super in its state, I'll overwrite those initial values here, like glutenFreeFilterSet with the values I got here in my widget.

So with the currentFilters.

Because here and initState, we can use this special widget property to get access to the currentFilters.

And then here we can access filter.glutenFree, for example, to get the initial value for this variable.

We just have to add an exclamation mark here to tell Dart that this will not be null, which we know it won't because this is a required parameter.

So with that, we're initializing it like this.

And here we also don't need to call setState because in initState will run before the build method executes anyways.

So these here are the values that will be available once the build method executes without us calling setState here.

And I'll then repeat this for all these state variables.

Here I'm accessing currentFilters, Filter.lactoseFree with an exclamation mark.

Then we repeat this for vegetarianFilterSet and access Filter.vegetarian.

And last but not least here we have veganFilterSet and access Filter.vegan.

So with that, we now initialize these state variables

to the actual values that have been saved
as state before.

At least if we make sure that the `currentFilters` are passed
to the `filtersScreen` when it's being activated.

And therefore we have to go to the place
where we are activating this screen.

So where we are pushing this screen,
which is the `tabs.dart` file,
and are in the place where we are pushing this screen.

Here, we now have to set
this newly added `currentFilters` parameter
and set it to our `selectedFilters` map.

And now we have to remove `const` here
because now dynamic value is passed to `filtersScreen`.

So with that, we now should preserve those filters
that have been selected.

And hence if we reload this
and I set some filters here,
those of course still affect those meal lists as before.

But if I go back to filters,
you now see that this was saved.

And therefore of course it also still applies
if I go back to my meals.

If I then go back and clear those filters
or choose different filters,
that of course will also be taken into account.

And now I see more meals again.

And that's now the finished meals app.

27. An Alternative Navigation Pattern: Using Named Routes

In this section, you learned about the recommended navigation approach: Pushing and popping `MaterialPageRoute` objects to load different screens.

There also exists an alternative approach, though - you can assign names to routes and load your routes via those names. However, that approach is not recommended (for most apps).

Nonetheless, you can explore this alternative approach via the official documentation: <https://docs.flutter.dev/development/ui/navigation#using-named-routes>

28. Module Summary

And that's it for this course section.

We built this meals app where we can dive into meals,
mark them as favorites,
and then also see them here under this favorites tab.

We can go to filters,
and they're set different filters,
which actually won't affect our favorites,
but which will affect what we see once we pick a category.

And whilst building this app,
you learned a lot about navigation
in multi-screen Flutter apps.

You learned that you can remove screens with pop,
but even more importantly,
you learned that you can add new screens,
and navigate to a new screen with the navigator object,
and then the push method.

Or also with the push replacement method,
which could be used to replace the currently active screen
with a new screen.

Because you also learned that in the end,
this whole navigation concept
is all about having this stack of screens,
where you push new screens onto this stack,
or pop them off that screen,
and it's the top most screen that's visible to the user.

That's how navigation works in Flutter apps.

And in this course section here, you did not just learn

how you can go forward and backwards,
and how you can dive deeper and deeper into screens,
but you also learned how you can add tabs
with the bottom navigation bar.

How you can store which tab is currently selected
to make sure that the correct tab is highlighted.

And how you can move between different pages
or screens based on which tab was selected.

And you also learned how you can add a side drawer,
so a drawer like this, which allows you to move
to different app areas, so to different screens
through that drawer based navigation.

You learn how that works,
how you can build your own side drawer,
and configure it exactly as needed.

And how you can also go back, and even make sure
that you got full control over this process of going back,
not just when using a drawer, but anywhere in your app.

Here we just used it in the context of a screen,
which we reached through a drawer,
but this would work anywhere.

So you'll learn how you can control this,
and how you can also return data when popping a screen.

Data that can then be used in the place
where the screen was pushed, like we're doing it here,
where we are using the result
to update our locally stored selected filters.

So we learned a lot of navigation related things
in this course section, but you of course also saw
that passing all this data, and all these functions
that then can be called from some nested widgets
to update some data, that passing all these things around

can be rather complex and annoying.

And that's the kind of problem we're going
to tackle in the next course section.

8_meals-app_managing-app-wide-state

1. Module Introduction

In the previous course section, we learned how to build multi-screen apps with Flutter and how to navigate between screens by building this app here, this meals app, where we can mark and view favorites and where we can also set filters to only view certain meals when browsing these meal categories. However, whilst building this application in the previous section, we also saw that managing state that affects different screen widgets can be quite cumbersome and that's exactly where this course section here will continue therefore. You will learn how you can manage application-wide or cross-widget state with ease without passing data around like crazy as we had to do it in the previous course section. For that, in this course section here, we are first of all going to repeat what exactly is wrong with the current solution or what the problem is with the current solution before we then explore a better solution, a third-party package called riverpod, which simplifies cross-widget state management a lot. And of course you're then also going to learn how to use that package, and by the end of this section, you will see that managing state

in complex applications actually doesn't have to be complex.

2. What's The Problem?

So what's the problem right now?

Well, at the moment in this app
which we built in the previous section,
we of course have a bunch of screens
which are related to each other
and between which we can navigate.

The problem is that, for example, in the favorites screen,
we need a list of favorite meals.

So this favorite meals state in the end,
we need that in that favorite screen,
but we change that state in the end
from inside the meal details screen,
because it's in this meal details screen
where we have this button to mark
or unmark an item as a favorite,
but we need the information in a totally different place
in that favorites list here.

So in theory, we would have to communicate
between these two screens,
but that's not possible like this at the moment
because we have no direct connection between these screens
because we reach the favorites screen
from the tabs screen, as you can see here,
whilst we reach the meal details screen
from the meals screen.

Therefore, to work around that,
we in the end have to manage this favorites state
in the tabs screen
and then there create a method that edits this state.

But since this method in the end needs to be called from inside the meal details screen, we have to pass it down to that screen through all these screen layers, and we're doing this by accepting this on toggle favorite parameter in all those widgets that are between the tabs screen and the meal details screen, and then there in the meal details screen, we can finally use this method. And whilst this approach, of course, works and also isn't necessarily wrong, it's, of course, inconvenient, and the more complex your applications become, the more complex this kind of state management will become. That's why we're going to explore a better solution in this section here.

3. Installing the Solution: Riverpod

Now this better solution, we're going to explore is the Riverpod package, a third-party package which you can use to manage cross-widget state in Flutter applications. Now, I will say right away that it's not the only way of managing cross-widget states, and there are many other packages as well. For example, also the Provider package, which is also very popular, but which was created by the same creators as Riverpod. And Riverpod simply is the more modern and better version of that Provider package, which is why we're using this one here in this course.

But there are different alternatives,
which you definitely also might want to check out.
But here, we're going to use Riverpod.
And Riverpod is a package
that is all about simplifying the process
of managing cross-widget state.
So therefore, to use it, we first of all have to install it.
And to install it, we have to use this package name
and we can then run `flutter pub add`
and then this package name in our project folder
in the terminal.
So this now adds Flutter Riverpod,
as a dependency as a third-party package,
to this project.
And with that, we can now use this package here.

4. How State Management with Riverpod Works

So how does this Riverpod package work then?
Well, in your Flutter app,
with that Riverpod package installed,
you can create a so-called provider,
which is some kind of object created
based on a class that is provided by the Riverpod package.
Now, this provider then can provide
a potentially dynamic value,
and it can also provide methods
that may change that value in the end.
And then in your application, in any of your widgets,
not just your screen widgets, but any widgets,
you can set up a consumer that is connected
to the provider and it will be connected automatically
by the Riverpod package.

And in that consumer widget,
you can then listen to changes to that provider value
or even trigger those changes by calling those methods
that might be provided by the provider.
That's how it works.
And because you have this central provider
and any widget can set up a consumer,
you don't need to pass these cross-state management values
between those widgets anymore.
Instead, every widget can directly connect to the provider,
and that's what we're now going to implement.
First with a basic example,
and then step-by-step with more complex examples.

5. Creating a Provider

So let's use this Riverpod package
in this project here,
and this is the same project as before, as mentioned,
where I installed this Riverpod package.
And in this lib folder here,
I'll add a new providers folder
where I wanna set up these globally available
provider objects.
Now in there I'll add a meals provider,
and I'll name this file meals_provider.dart
to make it very clear
that this file will contain such a provider.
Now, in this newly added meals_provider.dart file,
we should start by adding an import,
and we should import from the flutter_riverpod package,
and there the flutter_riverpod.dart file.
So that this import is added.

With that import added, we can add a provider by using this provider clause that is provided by flutter_riverpod.

Now instantiating, this provider clause will create such a provider object, which we then can listen to from inside our widgets.

And we should, therefore, store this object in some variable so that we do store this object and we can access this object later.

So for that, I'll add a new final variable called meals provider, and I'll name it mealsProvider because I plan on providing my meals data here.

And later, we're also going to add more complex providers, but we'll keep it simple here.

And the provider clause then actually takes a couple of parameters, which you can set.

But most importantly, it needs at least one parameter, one positional parameter, and that parameter actually is of type function.

It's a function which will receive a so-called ProviderRef object.

So a function that will receive an object automatically because this function, which we must pass to provider, will be called by the flutter_riverpod package.

And this function will receive such a ref, as mentioned, which is of type ProviderRef.

And inside of this function, we now should return the value that we wanna provide.

And here to keep things simple for now, I'll provide my dummyMeals.

For that, you must make sure

that you also import `dummy_data.dart`.

And of course, providing those meals is a bit redundant

because the `dummyMeals` is really just a list here,

which we could import in other files anyways.

And that is what we have been doing thus far.

So we don't really need to use this provider approach here.

We can't just import it.

The provider approach is more useful

if we have dynamic data and cross-widget state.

We will, of course, use that package

for this more complex state later.

But to get started, we'll provide the `dummyMeals`

even though it's technically not required.

But with that here, we now set up a provider.

And with that provider setup, we can now use it in widgets

that need the data that is provided by this provider.

So in this case, we can use this provider in widgets

that need our meals.

And in this application that would, for example,

be the `tabs.dart` file.

and there is the `TabsScreen`.

Because in this `tabs.dart` file

and they are in the `TabsScreen` state specifically,

I am using these `dummyMeals` in the `build` method

to derive my available meals

based on the filters that were set.

Now later, we're going to outsource this logic

into a provider as well.

But for the moment, I simply wanna make sure

that here I don't use the `dummyMeals`

that are imported from the `dummy_data.dart` file,

but that we instead remove this import.

And instead I use that provider, which I just created.

6. Using a Provider

Now to use this provider here

in this tabs.dart file, we first of all have to import it

by referring to our own package

and then the provider's folder,

and there the meals_provider.dart file.

And with that import added,

we also have to add another import

where we import this Flutter_Riverpod package here

and then the Flutter_Riverpod.dart file.

And with this import added as well,

we have to replace stateful widget

with ConsumerStatefulWidget,

which is basically a stateful widget

provided by this Riverpod package,

which has some added functionalities that allow us

to listen to our providers

and to changes in those provider values.

Now if we did not have a stateful widget here,

but a stateless widget, we would replace this

with ConsumerWidget like this.

But since we had a stateful widget here,

it's ConsumerStatefulWidget, which we'll need to use.

But we'll see consumer widget in action later as well.

Now with the widget class extending ConsumerStatefulWidget,

we also have to change the state class

also to get rid of this error here,

and here, we now have to extend ConsumerState

instead of just state.

In addition, we should also use ConsumerState

as a type here.

So now we got rid of all these errors,
and we're extending these consumer classes
for both the widget class and the state class.
Now by doing so, we behind the scenes add functionality
to those classes that allows us
to reach out to the providers we might have defined
like this meals provider here.

And we can use this functionality here
in the TabsScreenState class now.

There I want to get my meals,
and previously here I imported and used dummy meals.
Now we can get those meals from the provider.

And for that,
we've now got a new ref property available here
in this state class.

Just as we have widget available
to reach out to the widget that belongs to this state,
we now have a ref property available
that allows us to set up listeners to our providers.

So this ref property is added by Riverpod,
and it is available
because we are extending consumer state here
in this state class.

Now with this ref property available,
the question of course is what we can do with it.

And what we can do
is we can use a couple of utility methods.
Most importantly read to get data from our provider once
or watch to set up a listener
that will make sure that the build method executes again
as our data changes.

And the official Riverpod documentation recommends to use watch as often as possible, even if you technically only need to read data once, because this way if you ever change your logic, you can't run into unintended bugs where you forget to replace a read with a watch. That's why they recommend that you always use watch right from the start. Now watch needs a parameter though, and the parameter it needs or the argument it needs is a provider. So here we now wanna use the meals provider that we created and stored in this meals_provider.dart file, in this file here. We're defining it here. We are importing this file here in the tabs.dart file. Hence we can access this variable here, and we're passing it as a value to watch. Now with that, we set up a listener that will re-execute this build method whenever this meals provider should change. So whenever the data in there should change, which this data here of course never does, but this will change later when we build other providers. Now nonetheless, with that we got this listener. And in addition, watch also returns the data of the provider it watches. So here we also get our meals, which we can store in a variable. And you can see that it even infers the type correctly because it is able to infer the type of data we are returning here in our provider.

So with that, we now get our meals, which we can use here
to then filter those meals based on the filters we set.

With that, if we save everything and force a restart,
you get an error though,
and you get an error because creating a provider
and extending these consumer classes
and then adding a watch like this is important,
but it's not everything.

Instead, there's another very important thing
you have to do in your Flutter app if you wanna use
this Riverpod package.

You have to go to your main dart file,
and there, where you're calling Run app,
you have to wrap your app with another widget,
another widget provided by the Riverpod package.

Therefore here, we should also import
package:flutter_riverpod and then Flutter_riverpod.dart.

And with that import added, we should wrap this here
with another widget, which can also be set to const.

And that widget would be the provider scope widget,
which is a widget provided by Flutter_Riverpod.

Provider Scope takes a child parameter,
and the value of that parameter is our main entry widget,
so the app widget in this case.

But wrapping this app with provider scope
is important to unlock this
behind the scenes state management functionality in the end.

And we are wrapping the entire app here
so that all the widgets in the entire app
can use these Riverpod features.

If you knew that only a certain nested part of your app
needed those features, it would be enough to wrap that part.

But here I want to be able to use Riverpod
in the entire app, therefore I will wrap the entire app.
And with that now, if you save this and force a reload,
the error goes away.
And now I can still see my meals here,
once I select a category.
But now technically those meals are not fetched
by importing the dummy data file, at least not in a widget,
but instead they are fetched through that meals provider.
And that's how you generally set up
and then also use such a provider
by extending these consumer classes in the widgets
where you wanna use it.
And by then using `ref.watch`.
Now that was the basic use case.
Basic because we returned some dummy data,
which actually never changes.
Well, that's going to change now
because next we're going to build a more dynamic provider.

7. Creating a More Complex Provider with StateNotifier

So now that we got started
with the Riverpod package
and we built our first provider,
it's time for a second provider,
which I'll store in a new file
to keep things organized and separated.
And this will be my favorites provider,
which I wanna store in this file,
hence I'll name the file: "favorites provider."
And the idea here is to build a provider
that simply stores all these favorite meals

in a list of favorite meals.

For this, in this favorites provider dart file,
we of course need to import the Flutter Riverpod package
and the Flatter Riverpod dot dart file from that package.
And we also need to create a new variable,
the favorite meals provider variable, for example,
that stores a provider.

However, this time,
we're not going to create the provider like this
because this basic provider class here
is great if you have static dummy data, like here,
a list that never changes.

If you have more complex data
that should change under certain circumstances,
the provider class is the wrong choice.

In that case,
you should instead use this
state notifier provider class,
which is another provider class provided by Riverpod
that is optimized for data
that can change.

Now this state notifier provider class
works together with another class,
just as stateful widgets work together
with state objects you could say.

And that other class
with which the state notifier provider works together
is the state notifier class,
which is also provided by Riverpod.

Now this class is actually not instantiated
like this though.

Instead, you use this class to build your own class

that extends this class.

Now your own class can technically have any name you want,

but the convention is to end the name with notifier

and I'll therefore name my class favorite meals

notifier like this and extend state notifier.

Now, state notifier is actually a generic class

just as the built-in state class,

which is provided by Flutter is generic.

And here I'll therefore add angle brackets

to in the end tell Flutter Riverpod,

which kind of data will be managed by this notifier

and therefore ultimately by this provider, you could say.

And the data I plan on managing here

will be a list of meals.

Now, for a meal to be available,

we must make sure that we add an import

to the meal.dart file from the models folder.

And now with that, we can fill this class with some life.

And the life I do want to add to this class

is a, the initial value.

So in this case, for example,

initially an empty list of meals,

and B, all the methods that should exist

to change that value, to change that list in this case.

Now the initial data is set up

by adding a constructor function, which as you learned,

has the same name as the class.

And then here, we don't pass anything

to this constructor function

or we don't expect any parameters to be precise.

Instead, we add such a initializer list by adding a colon

after the constructor function parameter list.

And then here, we call Super
to reach out to the parent class,
so to the state notifier
and to Super, we pass our initial data.
Now that data can be of any shape of your choice.
It can be a list, it can be a map,
it can be another object based on another class,
it can be a string, but of course, it should be of that type
that you define here between the angle brackets.
So here, it should be a list of meals.
And therefore here I'll pass such a list
though at the beginning, it will just be an empty list.
With that, we set the initial data
that will be stored in this notifier
and therefore ultimately, also soon in this provider.
Next, we also add any methods we want to this class
that allow us to edit this data.
And here, this will just be one method,
which I'll call toggle meal favorite status,
though the name is up to you,
which expects to get a meal of type meal as an input,
and which then should add that meal
to that list if it's not a favorite yet,
and remove it otherwise.
Now for that, you could think
that you simply used the logic
we used before in the tabs widget.
There, we also got a meal
and then we simply checked
if it was already part of the favorites.
And if it was, we removed it with the remove method
and otherwise we added it with the add method.

This will not work here anymore
because one important thing you must keep in mind
related to that value that's managed by this notifier
is that you must never edit that value.

And you might recall from earlier in the course
that in the end when you create objects,
those objects are stored in memory
and they can be replaced
with new objects which have a new address in memory,
in which case the old object,
if it's not used anywhere else, would get deleted.

But you can also edit an existing object in memory
so that the address stays the same
and you manipulate that existing object in memory
instead of replacing it or instead of adding a new one.

And this approach is not allowed when using state notifier.
You're not allowed to edit an existing value in memory,
instead, you must always create a new one.

That's simply a pattern enforced by this package.

Therefore, we're not allowed to reach out to this list
and call add or remove as we did before,
this would not be allowed.

Instead we have to replace it.

Now to replace it,
there is this globally available state property.

This property is made available by the state notifier class.

Now this state property holds your data,
so in this case, a list of meals.

And again, calling add or remove on that
would not be allowed.

Instead, you have to reassign it
by using the assignment operator to a new list.

So a new list, no matter if you are adding a meal
or removing a meal.

Therefore, what we have to do here is we first of all
have to find out whether a meal is part of that list or not,
and then thereafter,
we have to create a new list based on that information.

Now, for that, we can of course, again,
use state contains meal though,
because that will not edit the state,
it will just look into the state,
and that of course is allowed.

And this will tell us whether a meal is a favorite or not.
So that's how we get this information.

But then if a meal is a favorite,
we'll have to remove it from that list
without using the remove method.

And this can be achieved by setting state equal to
state where, because remember,
with where, we filter a list and we get a new list
or a new iterable, which we can convert to a list
by calling toList.

So where gives us a new list
and that's of course exactly what we need here, a new list.

So where then gets a function
where we get the current list items, item after item,
and we have to return true to keep it
or false to drop it.

And here I want to keep a meal if its ID is not equal
to the ID of the meal I'm getting here.

Now here I would have a name clash effectively
because I have a bunch of meals here.

So to set those meals apart, I'll name my meal here just M.

And now this meal

refers to the meal I'm getting here as an argument.

So now if the IDs are not equal,

I know that the meal I am taking a look at

is not the meal I am toggling the favorite status for,

and therefore, I definitely want to keep that meal,

that's why I'm checking for inequality

because I want to get true if the IDs do not match.

Because on the other hand, if they do match,

I know that this is the meal that should be removed

and therefore I want to get false as a response

to drop it.

That's what I'm doing here.

So that's how I can remove a meal.

For adding a meal, I can use a different syntax

and set my state equal to a new list

where I simply want to keep all the existing items,

but then also add a new one.

And to do this, we can use the spread operator,

which you also saw earlier in the course already.

You can use it on a list.

So in this case, on state, which is a list of meals

to pull out all the elements that are stored in that list

and add them as individual elements to this new list.

And then separated by a comma,

we can add yet another new element,

and that would be this meal here.

So that we pull out and keep all the existing meals

and add them to a new list.

And we also add the new meal to that list.

That's what we're doing here.

And with that, we're updating this state in an immutable way.

So without mutating,
without editing the existing state in memory.
And that was a lot of talking about this,
but it is important to understand how we do store and access
and update this state
when using Riverpod's state notifier here.

So with that, we finished this
favorite meals notifier class, which has initial data
and a method to edit this data.

Now we just need to connect it to this actual provider
so that we can use this data in our widgets.

Now, to connect this,
we have to go to state notifier provider here
and to state notifier provider,
we still pass a function where we still get a ref
that does not change compared to the meals provider.

And in this function here, I now still wanna return a value,
so that also doesn't change.

But the value which I do provide here
is now an instance of this notifier.

So we use this class name favorite meals notifier
and instantiate this.

And now, this provider returns an instance
of our notifier class so that we have this class
for editing the state and for retrieving the state.

Now here, we should also do one more thing,
a state notifier provider is a generic type
and it does understand
that it returns a favorite meals notifier,
but it does not understand unfortunately,
which data this favorite meals notifier
will yield in the end.

Therefore, here we should add angle brackets
and add two generic type definitions here.
The first one is favorite meals notifier.
But the second one then is the data that will be yielded
by the favorite meals notifier in the end.
And here, that will be a list of meals.
So we should add this to get better type support
in the rest of our application.
Well, and with that,
finally this favorite meals notifier is set up.
This is how we would do it
if we would be using Riverpod for this,
and now we can use it in our widgets.
We can use it to get our list of favorites,
but then also to trigger this method
and change our favorites.

8. Using the FavoritesProvider

So, how do we now use this provider here?
Well, generally in the same way as before.
And for that we can, for example,
go to the tabs.dart file again.
Because there, in the build method of the tabs state,
we are rendering the MealsScreen
as our Favorite screen in the end, if that tab is selected.
And here, we're passing our favorite meals
to this MealsScreen.
Now, therefore it's here where we actually wanna reach out
to our favorite meals provider
to get those favorite meals
to pass them to the MealsScreen.
Now, you could argue that,

of course, we might wanna do this
from inside the MealsScreen instead,
because that is where we need those meals after all.
But keep in mind that this MealsScreen
is not just used as a Favorite screen,
but also inside of the CategoriesScreen
once we selected a category.
There, we also go to the MealsScreen.
And there, we then don't just wanna show the Favorites,
but all meals that fit this category.
Therefore, here, it actually makes sense
to keep the MealsScreen configurable
and accept the meals
that should be displayed there as a parameter
so that this MealsScreen widget
can be reused from inside different screens
and we instead get the favoriteMeals
in the place where we need.
Exactly, the favoriteMeals.
And that would be here if this tab is selected
in our tabs screen in the end.
Now, here, conveniently,
we are already extending ConsumerState
and we need to do this here
because we wanna access a provider.
We now just have to make sure
that we also do import this newly added provider,
so this favorites_provider here,
this import must be added to the tabs.dart file.
And with that added, we can go back to the build method,
and there, we can now also set up another watcher
here where we need our favoriteMeals.

And here we'll then get our favoriteMeals
by using ref.watch
and watching the favoriteMealsProvider.

(graphic whooshes)

So, this provider,
which we just defined here
in the FavoritesProvider Dart file.

We're watching it here,
we're getting the favoriteMeals here,
and it's now these favoriteMeals
which we pass as a value to meals.

This also means that we can get rid
of this favoriteMeals state variable here
in the tabsScreen state.

We can get rid of that,
and we can get rid of this entire method here,
because we are no longer managing
this list of favorites here.

We are instead managing this in our state notifier
and they offer in our provider.

Now, regarding this info message that we might wanna show,
I'll get back to this soon.

For the moment, I'll just keep it here,
and we'll move it somewhere else soon.

But now with that down here, we of course,
also have to do some cleanup work.

Because this toggleMealFavoriteStatus,
which we passed through all these layers of widgets,
can and should now finally be removed,
because now finally, we can get rid of this chain here,
and instead use our Provider
to access and change the data directly from the place

where we need to access or change it.

Hence, we should get rid of

the `onToggleFavorite` parameter here on the `CategoriesScreen`

and on `MealsScreen`.

And of course, it's not enough to just remove it here

in the place where we use these widgets.

Instead, we also have to go to these widgets

and there in the `CategoriesScreen`,

we should get rid of `onToggleFavorite`,

and also get rid of this property.

And then, of course, with that,

also get rid of the property here on the `MealsScreen`.

And then in the `MealsScreen`,

we also have to remove this parameter and this property.

And then here, we passed it

to the `MealDetailsScreen` in the past,

and this now also thankfully, can be removed,

making our overall code simpler.

In the `meal_details` screen,

we therefore now also have to get rid of `onToggleFavorite`,

because we no longer have to pass this function around.

And instead now,

we want to trigger this favorite status update,

which we defined in our favorites provider

or in the state notifier there.

To be precise, from inside this `MealDetailsScreen`.

And that's there for now, the next step.

9. Triggering a Notifier Method

So how do we now trigger this toggle meal

favorite status method in the favorite meals notifier

from inside the meal detail start file.

Well there as a first step we also must import
the Flutter River Pod package
and the Flutter River Pod dot dart file
and replace stateless widget with consumer widget.
Not consumer stateful, but just consumer widget.
With that done, the build method here has to change
because here we now get a second argument,
a second parameter passed in automatically by River Pod
and that is a parameter of type widget ref,
which is this ref thing which we need
for listening to providers.

We did not have to adjust the build method
in the tabs dart file because there we had a state clause
and we replaced that with consumer state,
which made this ref property globally available
in this class.

Here for the state less widget it's different,
we don't have such a general ref property,
instead we have to add it here
as a parameter to the build method.

But with it added here,
we can now use it to call methods
from our notifier that is exposed through the provider.

For that here in on pressed on this icon button,
we can use ref and now here in a change listener,
so in a function that is passed as a value
to on pressed or something similar,
you should not use watch but instead read
to not set up an ongoing listener
because inside of such a function that would be problematic,
but instead to just read a value once.
Now the value that should be read here

is again our provider.

So for that we refer to our own package
to the provider's folder
and there to the favorites provider dart file.

With that import added here,
we can use this favorite meals provider,
but now we don't use this value like this,
but instead since this is this special state
notifier provider which we created here,
we can access dot notifier here and this gives us access
to this notifier class we defined.

So to this class here,
which of course has this method we want to call.

That's why we're accessing this notifier here
because with that done on the result returned by Reed,
we can call toggle meal favorite status.

And to that method we can pass the meal,
which is available here in this widget class as a property.

And that is how we now trigger this method in our provider
from inside of this meal details screen.

And that's how we skip this long chain of passing functions
around because now we can trigger the change
from exactly the place where it happens.

Therefore, we can now also go back to the tabs screen
and copy this scaffold logic here
and remove this show info message method
into this meal details screen.

Here besides triggering this method,
we can also now use Scaffold Messenger
to clear snack bars and to show a new snack bar
and here the text I then wanna show
depends on whether this was added as a provider or not,

and to find out whether it was added or not
we can tweak this toggle meal favorite status method
in the favorites provider dart file
and instead of returning nothing,
we can return a boolean here where we may
be return false if the item was removed
and true if it was added.

With that we're returning some data from this method
and therefore in the place where we are calling this method,
which is this meal detail screen,
we now get this as a result,
this information whether it was added or not
and we can then set this text here dynamically
by checking was added.

And then maybe with a ternary expression
we can define what should be output if it was added
and what should alternatively be output
if it was removed basically as a favorite.

So here we could say meal added as a favorite
and then here we say meal removed,
something like this.

And with that we're triggering all these actions
from inside the place where they actually occur,
which of course is better than what we did before.

So with that, if we now save everything,
we should be able to navigate to a meal,
click this button and we still get the snack bar here,
but now as I navigate away,
I see this here and I can also remove it
and it's then updated and this all now works
without passing functions or data around,
which is of course very convenient

and much better than what we had before
because ultimately it leads to a leaner app.
And that's the huge advantage of this provider pattern
and of using a global state management solution
like this one here.

10. Getting Started with Another Provider

So we added this favoriteMealsNotifier and
therefore in the end, the favoriteMealsProvider.
And with that, we see a more complex example for a provider
that does not just provide static data, which doesn't change
as we're doing it with the meals provider,
but instead for a provider that does in the end,
provide an object that does allow us to change data.
And that typically is the more common use case.
And it's the same approach I now wanna use
for adding a filters provider though, for this provider,
for the filters provider, which we're about to add
in the next lectures, you will see
that we'll actually take it a step further
and build a provider
that depends on another provider as you will see.
So let's get started with that.
And for that, I'll add a new file in the provider's folder,
the filters_provider.dart file.
And in that file, the goal of course is
to add such a filters provider.
So therefore, we must import flutter_riverpod again
as we did it before in the other provider files.
And now in this file here, I in the end want
to create a filtersProvider
and store it in a variable or a final variable here.

And I will do this again

with the StateNotifierProvider just as we did it
for the favorites provider.

There, we also use the StateNotifierProvider.

Now I'll use the same provider here
for implementing the filtersProvider.

Therefore, we also must add a class again as we did it
for the favorites provider.

So here I'll add my FiltersNotifier,
which extends stateNotifier
and stateNotifier as before, is a generic class,
which in the end will manage a Map, a Map of filters.
And for that, this Map of course, should have Filter keys
and then Boolean values.

Now for the filter keys I did before,
provide them here in filter start.

Now I'll actually move this enum here from filter start
to the filters_provider file so
that all the meta management of filters
and all the type definitions
for my filters basically happen here.

So with that I have my enum here, I have my filter here,
therefore in this map definition.

And of course, to use this Filter type in other files,
we'll then have to make sure
that we import this filters_provider.dart file
into these other files.

But that is something I'll do later.

So with that here, I have this FiltersNotifier.

And then as before, for the FavoritesNotifier,
we must set an initial state
by adding the constructor function

and then the initializer list, where we call super
and where we then pass our initial state
to the parent class.

Now for the favoritesProvider,
the initial state was an empty list
for the filtersProvider, it should be a map of filters
where every filter is set to false initially.

So here, we basically have Filter.glutenFree set to false,
Filter.lactoseFree set to false,
and the same for vegetarian.

And then, of course, also for the vegan filter.

So that's now my initial filter state here.

And with that state-defined, we can then add a method
that allows us to manipulate
that state in an immutable way as you learned.

So therefore here, I'll add a method,
which I'll call a setFilter, where I expect
to get the filter that should be set as an input.

So a value of type filter
to identify which filter should be set to a different value.

And then I get a Boolean value isActive,
could be a fitting name,
which defines whoever this filter is now set
to true or false.

And with that data received, we can now update the state.

Now again, as mentioned before,
we must do this in an immutable way.

So accessing state filter like this
and setting it to the updated value would not be allowed
because here, we are mutating the state in memory
and that's not allowed.

Therefore, what you should do instead

is you should set the state to a new map
and that should in the end be your old map
with the updated key.

And one way of creating such a new map here is
to copy the existing map
and its existing key value pairs with the spread operator,
just as we did it for the list.

This also works for maps inside of maps.
So this copies the existing maps key value pairs
into this new map.

And then we can explicitly set one key like here,
this filter to a new value isActive.

So that will then override the key value pair
with the same filter identifier that has been copied
and all the other key value pairs will be kept along
with this new setting here.

And with that, we are creating a new map
with the old key value pairs
and one new key value pair
that overwrites the respective old key value pair
for the same filter.

And that's essentially all the logic we have to add here
to update our filters
or to be precise to update one of our filters in this map.

Now with that, we got this filtersNotifier
and therefore now in this filtersProvider,
we wanna return an instance
of our filtersNotifier like this,
same logic as before with the favoritesNotifier.

And as before,
we should add these generic type annotations here
to state notifierProvider

to get better typing support in the places
where we use this provider.

Here, we should make it clear

that this will refer to the filtersNotifier,
which then in the end will resolve to a map
where we have filters and boolean values.

With that, we got the basic version of the filtersNotifier,
and we can now use that version in our code.

Now in the tabs.dart file, I for example,
must import filters provider to get access to this enum.

Since I now moved this enum from the filters screen file
to the filtersProvider file.

I can also remove the meal model import here by the way,
in the tabs.dart file

because we're not using anything from that file anymore.

But on the other hand, as mentioned, I should add my import
to providers filterProvider to make sure
that the filter enum is available in this file.

More importantly though, we have to go to filters.dart,
and in there, we should also add an import
to this filterProvider.

So to providers and then filtersProvider.

But we must also add an import to River Pod, so
to flutter_riverpod like this.

Because with this import added here,
we can now connect the FiltersScreen widget
to our provider and get access to that provider.

We do that as you learned by replacing the stateful widget
with the ConsumerStatefulWidget
provided by flutter_riverpod
and state with ConsumerState.

Also here when we define this type,

and with that done,
we can use this ref keyword here in FiltersScreenState
to set our filters.

11. Combining Local & Provider-managed State

And here we can now actually
do something interesting.
But before we do that,
let's simply connect the provider to this widget
as we did it before for the other providers.
So when we initialize our filters,
we no longer have to do this
with help of the current filters that are received here.
Instead, we can get rid of this here in our Filters screen
because now we can get this information
from the provider instead.
So therefore, here in its state, we can use ref
and then call a read.
And here I'm using read instead of watch
because initState is only executed once anyways.
So setting up a listener here doesn't make too much sense
because this code won't run again anyways
unless the widget was removed and re-added.
So here I will read and I will read the filtersProvider,
and from that filtersProvider by using read here,
I'll get my activeFilters
as they are stored in the provider or in my notifier,
I'll get my map here,
and we can then, of course, use that
to access the glutenFreeFilter.
And the same for the other filters for lactoseFree,
vegetarian and vegan.

We can use this data from the provider
to initialize our local state in this widget.
And I do have this local state here,
because I, of course, still wanna make sure
that as users tap these different list tiles,
these different switches in the list tiles,
I do reflect those changes on the screen.
That's why I'm still managing this state locally
in this widget as well,
so that these updates to the UI as users tap those switches
are still performed.
But I initialize my switches here, or my filters,
based on the data stored in the provider.
And I, of course, wanna update
the data stored in the provider
as I leave this screen.
So therefore here in this onWillPop function here,
I now wanna reach out to my provider by using ref here,
and here, since we're in an event handler,
we should also use read instead of watch
as mentioned before.
And here I also read my filtersProvider,
but in this case actually filtersProvider.notifier
to get direct access to this StateNotifier class,
because it's that class that has the setFilter method.
And here I now wanna set my filter, but which one?
Well, here when the user taps the Back button,
I actually wanna set all my filters.
So therefore, indeed we should go back to filters_provider
and maybe add a second method, setFilters,
which sets all the filters,
where I simply expect to get my map of filter bool mappings,

which could be called chosenFilters.

And then I simply set my state to those chosenFilters.

So I expect to get a map of filter settings

and I then use this new map to override my existing state.

With this method added here, we can go back to filters.dart,

and here we can now call setFilters,

and now pass our map of updated filters to this method.

So this map here, instead of passing it to pop,

we pass it to setFilters

to update our filters in the provider.

With that, we also don't need to call pop manually anymore.

Instead, we can just return true here

to allow Flutter to perform the pop action

it wanted to perform anyways.

And we don't need to do this manually anymore,

because the only reason we did it manually

was so that we could pass some data along,

but we're now not doing this anymore,

because we're using this provider approach instead.

Now with that, we are storing these filters

as we're leaving this page.

But keep in mind that I did now remove

this current filters input data,

which we previously used to initialize our filters.

Now we're doing this on the provider,

that's why I removed this property

and parameter from the FiltersScreen widget class here,

and therefore now in the place where I'm using this class.

So in the tabs.dart file,

I also have to get rid of this filter data,

which I wanted to pass to the screen.

And we now also can set this to const.

And we now also won't get back a result anymore after navigating users to the FiltersScreen. Because we're no longer returning any data here, we basically removed this code where we returned data through pop. So therefore here we're no longer getting a result. And, of course, therefore here, we no longer need to manage the selectedFilters at all, because we're now doing this in the provider. This property here can now be removed, which, of course, now also means that we have to do something with this logic here, because we no longer got this selectedFilters property here. Well, one thing we could do is, of course, we can use ref, which we already have here, because the tabs state here already depends on consumer state. And we can watch our filtersProvider here to get our activeFilters, because that's exactly what this provider manages. So we can use this activeFilters variable here, because there we do have access to the filters, because now we're getting the filter data from the provider. And with that, everything should work again. If we now save this and restart the app, if I go to my Filters screen, and I set some filters here and then go back, that looks good. Some items are definitely being filtered out whilst we can still see other meals, like this one. So that's how we could implement this filter logic with a provider.

But that's not all we can do.

Instead, we can leverage this Riverpod package
and this provider feature a bit more.

12. Outsourcing State Into The Provider

Now whilst our app here works as expected,
we can all take this a step further.

For example, we'll soon move this logic here
out of this file.

But before we do that, I'll go back to the filter start file
and there to the filters screen.

What we're doing here at the moment is the following.

We are reading our filters provider data here
and in its state to initialize our local state,
which we need to manage to reflect these changes
that are made to the filters as users click these switches.

This makes sense and works,
but we could also get rid of this local state
and simply manage everything in the provider
and simply update this filters screen widget
whenever data in the provider changes
to then update the switch status on the ui.

So put in our words, instead
of updating local state here when a switch is pressed,
we could simply update that filter in the provider
and then re-render the UI
to reflect those changes on the switch.

And that's what I wanna do next.

Therefore, to do that, I'll get rid of these properties
and actually of the entire state class
because now we won't need a stateful widget anymore
because now the goal is

to no longer manage any local state here.

Therefore, we can get rid of all of that.

And instead of extending consumer stateful widget,

we now just extend consumer widget.

So we can vastly simplify this here.

We can also get rid of in it state, again,

simplifying this all.

And now here I'll also get rid of this commented out code

to make this a bit leaner.

In the build method,

we must now accept this second argument widget ref

or of type widget ref

to be precise since we're now extending consumer widget.

So since we now essentially have a stateless widget here,

and with those changes made, we can also get rid

of will pop scope,

because now instead of only updating the filters

as the user navigates away,

I'll simply update them every time a switch is clicked.

So we can get rid of this

and directly return our column here instead, as a body

of course, we also must get rid

of the unnecessary closing parenthesis, therefore,

and now with all those changes made, we can go

to our switches like this one here.

And when this switch is pressed, I now want

to use my ref here to read my filters.

Provider state here, again, read,

because we're in such a event handler function here in the

end, a function passed to unchanged.

So here I read and then I access my notifier.

And then here I call set filter, not set filters,

but set filter because I wanna set a single filter.

And here that would be the gluten-free filter,
which I'll set to is checked.

So whatever value was selected by the user.

Of course I also must update the value now
because this local state is no longer available.

But of course we have this rev object here.

So at the beginning of the build method, we can use that
to watch our filters provider.

And here we want to use watch instead of read,
because watch sets up a listener
that Ree executes the build method
whenever the state in the provider changes.

So whenever a filter changes, for example,
and I want to re-execute the build method in that case
to update my switch tiles correctly.

So here I get my active filters, if you want
to call it like this, which is this map of filter data.

And then here, for the value of the gluten-free filter,
I can simply access active filters
for filter gluten-free like this
and add an exclamation mark to tell dart
that this will not be null, which I know it won't
because we'll always have a value inside
of our provider state.

So that's the first switch list tile,
and we can now use the same approach
for all those list tiles, copy, value, and unchanged.

And then here for the second switch list tile
for the lactose free one, I'll paste this in
and simply access lactose free and update lactose free.

And then copy

and paste this again here for the next one
for the vegetarian, this one,
and of course also update this accordingly.
And then do this for the last one here
where I access the vegan filter and update this one, vegan.
And with that, we made this widget,
this filters screen widget way leaner
because now we're outsourcing the state to the provider
and we can therefore use a state less widget effectively,
and we don't need to manage or initialize any local state.
As you will see, if you save everything
and restart the app, it still works
as before though, if I check gluten-free,
I still don't see the pasta here.
So that is working as it should.
My filters are saved and updated as expected,
but now all done through the provider.

13. Connecting Multiple Providers With Each Other (Dependent Providers)

Now, with that, we're almost done,
but there's one additional thing I wanna do
related to providers.
Because I mentioned before
that we're now also going to build a provider
that depends on other providers.
And I'll do this to get this logic here,
which we currently have
in the build method of my tab screen state,
to get this logic here out of this build method.
Instead, I wanna move this logic here
essentially into my filters_provider file.
And I'll do this by adding another provider here,

because of course,
you can add multiple providers to the same file.
I just typically like to spread them across multiple files
to keep things separated,
but if multiple providers are closely related to each other,
it also makes sense to put them into the same file.
And that's exactly what is the case here.
Because here, I'll add a new provider,
which I'll name `filteredMealsProvider`.
And as the name suggests,
here I wanna create a provider
which returns a list of filtered meals.
So, in the end, a provider that contains this logic here.
And therefore I can already cut this logic
and bring it over to `filters_provider.dart`.
Because here, this provider will now be a simple provider,
not a `StateNotifierProvider`,
because I won't create a new `FiltersNotifier` class
or anything like that.
But instead, a standard provider, which gets this function,
which receives a `ref`,
where in this function
I now paste in that code, which I copied,
where I then want to return this filtered list.
But of course, now `meals` isn't available here
and neither is `activeFilters`.
So, in the end, now this `filteredMealsProvider`
must depend on both the `filtersProvider`,
which of course may change,
because the filters that have been set can be changed.
But we also must depend on the `meals_provider`,
which indeed will never change,

but which in theory could also provide dynamic data.

So, how can we depend on these providers
inside of this provider?

Well, by using this ref argument here.

Up to this point, whenever we created a provider,
we never used this ref argument.

We only did that in our widgets.

But the ref argument we get here,
automatically by Riverpod,
is in the end, the same ref argument,
the same ref object we can use in our widgets.

It allows us to read or watch providers.

And that's exactly what I want to do here now.

Inside of this function, I wanna set up a watcher
to my mealsProvider, for example.

Now, for that, we must import this mealsProvider.dart file
from the provider's folder.

But with it imported, we can set up a watcher.

And if you do set up a watcher here,
the riverpod package will make sure
that this function here gets re-executed
whenever the watched value changes, so that in the end here,
you can then return an updated data
and any widgets that would be listening
to this provider would then also be updated if one
of the dependencies of that provider would update.

All of that would happen automatically behind the scenes.

So with that here I get my meals,
and now we can do the same for the active filters.

Here I can get my active filters
by watching my filters provider like this.

So this provider here, I'm watching this here,

so whatever this changes, this function is re executed and therefore the filtered meals are recalculated.

Well and now with this new provider added, a provider which now depends on other providers, which as you see is super easy to implement.

But with this provider added, we can now go back to the tabs.dart file, and there, instead of manually getting the meals and the filtersprovider, we now can get the available meals by using ref.watch.

And then here, I wanna watch my filteredMealsProvider. So this new provider I just added, which returns this list of meals, this list of filtered meals to be precise.

And with that, we moved this filtering logic out of this build method here.

This now also means that we can get rid of this meals provider import here because we're not using anything from this file here anymore.

But with all that, if we now save this and restart the app, you will see that if you set some filters here, this still works and certain values are still filtered out whilst other meals are still visible.

So this app still works as before, but now we're also using a provider that depends on other providers.

14. Swapping The "Favorite Button" Based On Provider State

Now with all these providers added, there's one last tiny change I wanna make to this app.

And this change is related to this Favorite button.

It, of course, works,

but now I wanna make sure that we swap this icon

as we change the favorite status of a meal.

And to achieve this, we can go to meal_details

because that is where we do have this icon button.

And here in the end, I wanna replace the icon

based on the current favorite status of this meal.

Now to do that here,

I'll use this ref object and set up a watcher

where I watch the favoriteMealsProvider that we created,

which returns this list of favoriteMeals.

And now I want to take a look at that list

and see if this meal,

for which this meal details screen here is being displayed,

is part of that list.

Now this can be achieved by accessing favoriteMeals.

And then there we can use

the good old contains method again

to check whether favoriteMeals contains this meal here

for which this screen widget is being rendered.

And then here we have the result,

which is a Boolean,

and I'll store it in a variable called isFavorite.

And this is either true or false.

And we can now use this Boolean

to render different icons down here.

For that, we have to remove const

because now we'll set the icon to a dynamic value.

And here we can now check

if this meal here is a favorite

by taking a look at the newly added isFavorite variable.

And if this is true,
using a ternary expression here,
if this is true, I'll display this star icon.
If it's false, I'll display the star_border icon,
which looks like this.
So an empty star, which I guess makes sense
if this meal is not a favorite.
And with this change made,
if we save this and maybe restart this,
you will see that if we dive into one of these meals,
initially this is an empty star,
but as I tap it, it gets swapped for the different icon.
And I can go back and forth.
And that, of course, then accurately reflects
whether this meal is a favorite or not.
And that's another tiny addition which I wanted to make,
which now is super easy to implement
thanks to this provider feature.

15. Module Summary

And that's it for this course section.
In this course section,
you learned about providers
with help of the Riverpod package,
which is just one of the many options you have
for managing global state or cross-widget state,
state that affects multiple widgets.
And as you learned,
packages like this can be very useful
because in more complex apps,
it's quite common that state changes
and state usage don't happen in the same widgets,

but instead might be far apart from each other.

And in such cases, without such solutions like Riverpod, you would have to pass functions around in your app so that they can be triggered from different areas in the app than the state is needed.

With Riverpod, you can instead set up a global state management provider, so to say, which provides a dynamic value and may also provide methods for changing that value, which then can be consumed from anywhere in your application so that you can read and change these values from exactly the places where you need to read and change them.

For that, you learned that after installing Flutter Riverpod, you must set it up here in the main.dart file by wrapping your app or whichever widgets need access to this provider feature with provider scope.

And then you can start setting up some providers, simple providers like this one here, the meals provider, which only returns some pretty static data, or more complex providers, which also allow you to change data.

The latter can be done by creating these notifier classes based on state notifier combined with the state notifier provider class.

Either way, you have these providers which then can be consumed in your widgets with help of the `ref` keyword, which is available if you extend consumer stateful widget and consumer state in case of stateful widgets

or simply consumer widget in case of stateless widgets.

In that case,

you would not have a globally available rev keyword,

but instead this second argument,

which you must accept in the build method,

which gives you this rev object,

and it's then this rev object which can be used

to set up listeners that make sure

that the build method is re-executed

whenever the provider data changes

and which then give you access to the provided data.

Or, as you also saw,

you can also read values one time to then, for example,

also trigger those methods

that may change the state by reaching out

to the notifier class that is related

to such a notifier provider.

That's how you can work with those providers.

As you learned, you can also set up providers

that depend on other providers by using this rev object

that's automatically passed to these functions

that you must accept when setting up a provider.

And with that, you now have a powerful

cross-widget state management tool

which makes sure that you don't have to pass data

and functions through all your widgets.

16. "riverpod" vs "provider" - There are many Alternatives!

"riverpod" vs "provider" - There are many Alternatives! Older versions of this course used the "provider" package instead of "riverpod" for app-wide state management.

riverpod is a library created by the same developer as the provider library - it's essentially a re-write of the provider package, fixing many of the flaws of that library (also see: <https://github.com/rrousselGit/riverpod>).

That's why this course uses riverpod.

As mentioned in the section, there generally are many other alternative packages you could use instead - for example Redux or Bloc. This [page](#) from the official documentation gives you a good overview of available packages - definitely feel free to play around with them and find the package you personally like most.

9_meals-app_adding-animation

1. Module Introduction

In this course section,
we are going to stick to the app
on which we worked over the previous couple of sections.
But in this course section,
we're going to enhance it with animations.
So we're going to make things move.
And you will learn how you can add animations
to widgets in Flutter,
and you will learn that you can either add explicit
or implicit animations.
And of course I will show you what the difference is
and how these different animations are created.
And we will therefore build a custom animation
from the ground up.
But you will also learn
how you can use various built-in animation widgets
to build and add animations with ease.

2. Setup & Understanding Explicit vs Implicit Animations

Now as mentioned,
we're going to stick to the project
on which we worked over the previous couple of sections.
Therefore, attached, you'll find my latest project snapshot
of the lib folder,
so you can simply drag and drop that lib folder
into a new Flutter project.
You also find my pubspec.yaml file attached,
and you should make sure that you add google_fonts,

transparent_image, and flutter_riverpod
to your project with their latest versions,
which can differ from these versions here.
But you should add these packages
so that you are ready to go.
And with that, we can dive right in.
Now, we will start by adding an explicit animation.
But of course, for that,
you first of all must understand what the difference
between explicit and implicit animations is.
And in the end, the difference is that,
with explicit animation,
you build and control the entire animation.
You have detailed control over its configuration,
whereas, with implicit animations,
Flutter controls the animation
and does a lot of work for you.
Therefore, with explicit animations, you have more control,
but also more work to do, and therefore more complexity.
And on the other hand, with implicit animations,
you have less control,
but therefore also less complexity to manage.
And indeed, often you can avoid building
your own custom animations.
So often you can avoid explicit animations
and instead use one of the many pre-built animation widgets
offered by Flutter to make your life easier
and add animations to widgets with ease.
Now in this section, we're going to see both,
but, as mentioned, we're going to start
by adding an explicit animation.

3. Explicit Animations: Adding an Animation Controller

So let's add an explicit animation.

And for that, I'll go to the categories screen

where I have this grid view

where I display a grid of category items.

So in the end, I'm talking about this screen here.

I wanna add a little animation to that screen

that lets these category boxes,

these category items,

slide in from the bottom

whenever we load this screen.

So when we, for example,

get to this screen

from the favorites screen,

or if we just started the app,

and such a slide animation could be built

by adding our own custom explicit animation

to this categories screen.

Now to do that,

we first of all have to convert

this categories screen class here

to a StatefulWidget.

And for that,

we can right-click on StatelessWidget

and choose Refactor

and choose Convert to StatefulWidget.

Because if you're adding an explicit animation

to a widget, you must add it

to the state object of that widget.

So you must be working

with a state object because,

behind the scenes,

an animation sets the state
and updates the UI all the time
as long as the animation is playing,
which is why it needs such a StatefulWidget
so that it can update some state
behind the scenes
and re-execute the build method.

That's the idea here.

Now with this changed,
you can go to your state class,
and here you first of all have to add something.

And in there,
you can now create a new property
that could be called `_animationController`.

The name is up to you.

But in this property,
we are ultimately going to store a value
of type `AnimationController`.

However, this `AnimationController`
can't be created here
where the property is initialized.

Instead, you should use `initState` for this.

So you should add this `initState` method
about which you already learned
earlier in the course.

And in this `initState` method,
you can then create this `AnimationController`.
Nonetheless, this `AnimationController` must be set
before the build method executes.

And to tell Dart
that this property doesn't have a value
initially when the class is created,

but will have one before it's really needed,
you can add a special keyword
in front of `AnimationController`,
which we haven't used up to this point.
You can use `late`, the `late` keyword,
and this tells Dart
that this in the end is a variable
which will have a value
as soon as it's being used the first time,
but not yet when the class is created.
So that there simply is a little timing difference
between when this property is created technically
and when it will have an initial value
and when it will be needed for the first time.
With `late`, you tell Dart
that it will be fine.
Once this property is used,
it will have a value.
Now speaking of a value,
you should still set the type of value
that will eventually be stored in here
after `late` before the property name.
And the type here will be `AnimationController`,
which is a type built into Flutter.
To be precise,
it's a class provided by Flutter.
And as you learned
in this course already,
every class brings its own type.
Now we can initialize this `AnimationController`
in `initState`,
so then it will be ready

before build is executed.

But as mentioned,

not right away when the class is created,

but that's fine because of late.

So here in initState,

we can now set _animationController

equal to AnimationController

using that class provided by Flutter

to create an object based on that class.

And now this class's constructor function

offers a bunch of configuration options here,

and there is at least one option, vsync,

which we must set.

Now, vsync wants a so-called TickerProvider.

In the end, this vsync parameter is responsible

for making sure

that this animation executes for every frame,

so typically 60 times per second

to overall provide a smooth animation.

For this, it must be set equal

to such a TickerProvider.

And this TickerProvider is actually received

by using another special feature

offered by Dart,

which we haven't used up to this point.

And which, to be honest,

you typically don't have to use that often,

but you do need it here

for explicit animations.

You should add the with keyword here

after extending this state.

And that's another keyword built into Dart,

a keyword which allows you to add
a so-called mixin to a class,
which you can think of another class
being merged into this class
behind the scenes
and offering certain features
to this class, therefore,
and the class that should be merged in here
is the `SingleTickerProviderStateMixin`
which is provided by Flutter.
And behind the scenes, this class,
which we're merging into our class here,
simply provides various features
that are needed
by Flutter's animation system.
By the way,
if you had multiple animation controllers
and not just one,
the right choice here
would be the `TickerProviderStateMixin` like this.
But if you only have a single animation controller,
it is recommended
to use the `SingleTickerProviderStateMixin` instead.
Now with this added,
the value for `vsync` should be set to this,
so to this entire class,
which now has this ticker thing merged into it.
And therefore, behind the scenes,
this `AnimationController` will be able
to get this frame rate information,
which it needs
to fire our animation once per frame.

So for example, 60 times per second.

That's what we're setting up here.

Now, in addition to vsync,

which is always required

when creating an AnimationController,

you can also add a duration

to control the duration of the animation

you want to play.

And for that,

here you can create a Duration object

by using Flutter's Duration class.

And here I'll set my duration to,

let's say, 300 milliseconds.

And of course, you can play around

with that value

to control how long this animation will play.

So this animation

where I wanna slide up my grid items here.

Now, besides the duration,

you can also set a lower bound,

for example, to zero,

and an upper bound.

And with that, you will in the end control

between which values Flutter will animate.

And this might sound strange,

but it's important to understand

that with animations,

you in the end always animate

between two values,

for example, zero and one.

And then it's your job

to translate these values into something

that changes something on the screen.

For example, you could be adding
some margin or padding to a widget,
and that margin or padding could be based
on the current animation value,
which is between these two boundaries.

And the AnimationController
will automatically move
between these boundaries
during this duration.

And if this is still a bit abstract,
you will see it in action
in just a second.

By the way, these settings here,
zero and one, are the defaults
and don't need to be set.

But you can of course also set them explicitly
if you want to or set them
because you want to animate
between different values like zero or 20.

Again, it will depend on what you plan to change
based on these values.

So here I'll stick to these values
and explicitly add them
even though they would be the default.

But now with that,
we have an AnimationController,
which we can use to play the animation.

However, that's not all we have to do.

Besides now using this controller,
which we'll do in a second,
we also should add this dispose method here,

which is another method
built into Flutter widgets,
which is called automatically
behind the scenes,
but which you can override
to then still call the parent widget's method
behind the scenes,
but to then also perform your own cleanup work.

And here I'm using this dispose
to use my AnimationController
and call dispose on that.

This makes sure
that this AnimationController is removed
from the device memory
once this widget here is removed
to make sure that we don't cause
any memory overflows or anything like this.
And now with that,
we're ready to use this AnimationController
to actually change some values
that will have a visual impact.

4. Explicit Animations: Playing the Animation with AnimatedBuilder

So how can we now use
this animation controller?
Well, I mentioned that behind the scenes
it would update the state
and kind of re-execute the build method therefore.
Now technically that's not entirely correct though,
because if it would do that,
it would mean that everything in the build method
is rebuilt and reevaluated 60 times per second.

If you have a frame rate of 60 frames,
and whilst you want to do that
for the things you're animating,
you might not want to do that for all the other things
that might be on the screen.

For this screen here, it actually doesn't matter
because here I do only return this grid view,
which I do want to animate.

There is nothing else here,
but for other screens it might matter.

So therefore, animated controller does actually not call
build multiple times,
but instead you can think of it as a timer
or an interval running behind the scenes
to which you can listen
to then manually update parts of the UI.

And you can listen and update parts of the UI
by using another widget provided by Flutter,
the Animated Builder widget.

This Animated Builder widget wants an animation
and a builder.

Now the animation is a listenable object,
and as it turns out, our animation controller
is such a listenable object.

So we can pass our animation controller as a value
for animation to Animated Builder,
and behind the scenes
this will basically tell Animated Builder
when it should call this builder function here,
because this function which we pass to builder,
this function here, will then be executed
for every tick of our animation.

So for example, 60 times per second.

And then inside of this function,

you can update a widget's padding or margin

or whichever value you wanna animate.

So for example here, I'll take my grid view like this

and cut it and use that as a value for this function here.

So as a return value returned by this function.

Now in this function you also get some arguments.

You get a context and a widget.

Now the context is something we know.

The widget we get here is an extra widget,

which we could add here on Animation Builder.

We could add an extra child parameter here

to set any widgets that maybe should be output

as part of the animated content,

but that should not be animated themselves.

And this would allow you to improve the performance

of an animation by making sure that not all the items

that are part of the animated item

are rebuilt and reevaluated

as long as the animation is running.

And indeed, this is a feature which we can use here

to actually set our grid view here as a child

by cutting it again and setting it as a child.

And by then adding a, let's say, padding widget here,

which is then the actual widget that will be animated,

which child is set to this child,

which is the grid view.

And therefore the grid view would technically be displayed

inside of the padding, but it would not be rebuilt

and reevaluated 60 times per second.

Only the padding widget would be rebuilt

and reevaluated that often.

And from a performance perspective,
that's of course preferable to re-executing everything
that often.

Now the padding widget also needs a padding value.

And here I now wanna set a value
that's dynamically animated.

And for this we can of course again use edge insets,
but now here I'll use edge insets only
to set the padding to the top.

And this padding value should now be set dynamically.
It should be set dynamically based on these bound values
between which I'm animating.

And the animation controller will start
at a value of zero and end after 300 milliseconds
at a value of one.

They offer to animate my padding here, my top padding.
I could set this equal to `animationController.value`,
which gives me access to the current value,
which will be between zero and one as you learned
times 100 for example.

Now we would animate up to a padding of 100
because initially animation controller value is zero,
and zero times 100 is zero.

But at the end, after 300 milliseconds,
we would have a value of one here,
and therefore the padding would be one.

Now if I do that and save everything,
we can therefore see if I go to favorites and back,
nothing happens.

And the reason for that is that indeed out of the box
just by adding such an animation controller,

the animation will not be played.

To play it, you have to explicitly start it.

And we can do this here in initState

right after creating it.

For this, you can target the animation controller again,

and then here you got various controls.

For example, you can stop it with stop

if you would want to pause it whilst it's playing,

for example, because a user tapped some widget

or anything like this.

And you can also start it by calling forward.

This will start the animation and play it to its end

unless you stop it in between.

You can also use repeat instead of forward

so that it would repeat the animation

and always restart it once it's done.

But here I'll go for forward.

And with that, if you save this and then restart this,

you will see that indeed if I go to favorites and back,

this animation plays, and it slides down these items,

but wait a second slides down?

I wanted to slide them up.

Well, we simply have to adjust our logic down there

in the Animated Builder.

Currently I'm animating from zero to 100 here

with this logic.

To slide my items down, we should change this logic,

and for example, set the padding to the top to 100

minus animation controller value times 100.

This will initially be zero and

therefore will start at a padding to the top of 100.

And by the end of the animation,

animation controller value will be one.

So this here will be 100 and 100 minus 100 will be zero.

So therefore, at the end of the animation,

the padding to the top will be zero.

And with that it should indeed slide up.

So if we save this ,now this looks better.

And now I got this animation as I switch to this screen.

But by the way, not if I come from the filters screen

because here this categories screen was never removed
from the stack of screens.

That's different with favorites.

And the animation, of course, only restarts

if the widget is re-added to the screen

because we started in initState,

and initState is only executed when the widget is created.

When we move to the filters screen,

the categories screen is still on the stack,

and it's not recreated if we come back to it.

It's just shown again.

That's why we don't get the animation here,

but why we do get it if we, for example,

come back from the favorites screen.

So that's how you can add explicit animations

by adding such a animation controller, configuring it,

starting the animation,

and then also controlling what should be animated,

for example, the padding.

And here as mentioned,

we're also using the performance optimization

of putting the content

that actually should only be part of the animation,

but which should not be rebuilt

because its values don't change
by putting that content into the separate child here
so that it's included in the animated content
but not rebuilt,
which helps us with improving the application's performance.

5. Finetuning Explicit Animations

Now, this animation here works
and doesn't look too bad, but we can also build it
in a slightly different and typically better way.
Instead of animating our padding like this here,
we can also return a different value here in our builder.
We can return a slide transition.
And indeed, Flutter has many transition widgets built in.
If you take a look here for example,
also have a FadeTransition, you have a SizeTransition
to animate between two sizes for a widget.
And AlignTransition to animate the position of a widget.
A ScaleTransition to scale a widget up or down,
or for example, a SlideTransition to animate the movement
of a widget from one position to another, so to say.
And for that, SlideTransition needs multiple parameters.
For example, it needs the child that should be animated.
In this case, it's still this child,
which we're outsourcing here, the grid view,
so that this grid view is not unnecessarily reevaluated.
But that's the child that will be animated
by slight transition in the end.
Now, position also must be set,
and that's actually not a number,
but instead an animation that animates an offset.
Now, what's that?

Now, an animation in the end is what we get
with help of animationController.
That's an animation in the end.
And offset then is a special kind of value
used by Flutter to describe the,
well, amount of offset of an element
from the actual position it would normally take.
And to make this a bit clearer,
let's simply start adding a value here.
And the animation I will add is my animation controller.
And on that controller,
you can call the built-in drive method.
Now, this simply is a method
that can be used to build an animation, as you can see,
based on some other value.
So in the end, it is a method that can be used
to translate this animation between these two values,
zero and 1 to an animation
between two other kinds of values.
For example, between two offsets here,
because here we need an animation over offsets,
not an animation between two doubles.
Not an animation between zero and 1.
And drive simply allows us to translate our animation
between zero and 1 to an animation
between two other values.
For that, drive wants an animatable child.
And such a animatable child can be built
with Flutter's Tween class.
The Tween class creates tween objects,
and the name tween simply comes from the fact
that this class and these objects are all about animating

or describing the transition between,

hence the name tween, two values.

And for that tween takes two values,

as you might guess, begin and end.

And now, here we wanna animate between two offsets

because position wants an animation over offset values.

Such a offset can be created with Flutter's Offset class.

And Offset simply takes two doubles,

two numbers with decimal places.

The first number describes the offset on the X axis

and the second number on the Y axis.

So, here where we're talking about sliding some element,

we're basically saying how much to the right

or left on the X axis and how much to the top

or bottom on the Y axis should we start?

And here you can enter values for both X and Y,

between zero and 1.

And zero simply means that there is no offset.

And 1 means that there is an offset by 100% in the end.

And here I want no offset on the X axis initially,

but I want some offset, let's say 30% offset on the Y axis,

so that the content is pushed 30% down.

And this can be a const value

because even though we're going to animate,

the starting point is constant.

And the same is true for the endpoint.

And here I want to have an offset of 0, 0

because I wanna end at the actual position

my grid view would normally have.

So, now these are the two values

between which we wanna animate.

And now, Flutter will figure out how to translate

this animation between zero and 1

to this translation between these two values.

That is what drive does in conjunction with Tween.

And with that, if we now save this and reload,

we will see that we again get a sliding animation.

But now actually, thanks to us using SlideTransition,

this animation behind the scenes is a bit more optimized

and it also gives us some extra features and capabilities.

So, which extra features do we get

by using this way of animating the SlideTransition?

Well, on this Tween, we can call the animate method

and animate all the returns and animation.

Therefore, we can indeed replace animationController.drive

with Tween animate since animate returns such a animation

over an offset, which is exactly what position wants.

But now, animate takes an argument

which we can use to get more control

over how the animation is played back.

To be precise, we can pass in a CurvedAnimation here.

Another object created with help

of another class provided by Flutter.

Now, CurvedAnimation in the end is a configuration object

for creating such animation

with help of this animate method here.

CurvedAnimation then takes a parent,

which is nothing else than our animationController.

So, the thing that controls the animation.

And it then takes a curve.

And the curve here wants a curve value.

But no worries.

Here, you don't have to create one yourselves.

Instead, there is a Curves object or a Curves class

built into Flutter

with a bunch of predefined animation curves.

And these curves simply control how the transition

between the begin and end state will be spread

over the available animation time.

So that we, for example, don't have a linear animation,

but one where we start slow and end slow

so that we have some acceleration in the middle,

but we end smoothly and start smoothly.

And you can simply play around with these different curves

to give your animations a more natural feel.

That's the idea behind setting curves.

And that's in the end the idea behind calling `animate` here

and passing such a `CurvedAnimation` configuration object.

It simply gives us more control

over how the animation feels.

If we save this and reload, you might see

that now the animation looks a bit more natural,

even though the difference, of course, is subtle.

But that's another way of explicitly setting

and controlling an animation.

And you can, of course, now use this knowledge

to animate all kinds of things in your Flutter app.

But as you will learn over the next lectures,

you often can also use implicit animations,

which can make your life easier.

Because with explicit animations, as you can see,

you have quite a lot of setup work to do.

With implicit animations as you will see,

you will have much less work to do.

So, let's take a look at those.

6. Getting Started with Implicit Animations

So now that we covered explicit animations,
let's dive into implicit animations.

And implicit animations
are all about using prebuilt widgets offered by flutter
that can be used to add animations
to certain areas of your app.

And for that, attached you'll also find a link
to the flutter documentation
to a page that lists some of the animated widgets
offered by flutter that can be used
to add such implicit animations.

You'll see an example
for one such animated widgets over the next minutes.

But you can also take a look at this list here
to find more widgets that could be helpful to you.

Now in this app here,
I wanna add an implicit animation
in the MealDetails dart file,
so on the MealDetailsScreen.

There we have this icon button in the app bar,
which we use to mark a meal as a favorite or not.

So in the end, this button here.

And now I wanna add a little animation to this button
so that whenever we click this button
and we change this icon,
we also animate the icon to, for example,
give it a spin and rotate it as it changes.

And whilst we could add this functionality
with an explicit animation
with our own animation controller,
it's even easier to add this kind of animation

by using an implicit animation.

So by using a prebuilt widget offered by flutter.

And indeed for many animations

you might want to perform pre-built widgets

are often the better choice.

So here for this MealDetailsScreen

for animating this icon here

whenever the favorite status changes,

we can go to this icon parameter.

So we really only want to go to the place

that should be animated.

I don't want to animate the entire button

if it's really just the icon that I want to animate

because the event listener here can't really be animated.

It's not visible.

So I'm going to this icon here.

And here, as a value for icon,

instead of setting this icon here as a value,

we can set animated switcher as a value.

And that is a widget provided by Flutter,

which allows us to animate the transition

from one widget to another.

So from this star icon

to this star border icon, for example.

By the way, I can't use animated switcher here

inside of icon

because animated switcher, in the end, is a widget.

And inside of this icon widget here,

Flutter wants an icon data value.

Icon, on the other hand, wants a widget,

so therefore, here I can use animated switcher.

But how does animated switcher work now?

Well, it has a couple of parameters that we can set.

For example, the duration of the animation,
but also the child.

And the child, in the end, should be our icon here.

So we can paste this in here

because that's the child which we want to animate.

Now we also have to add a duration here, of course.

And for the duration, you can create such a duration object
and it's up to you which duration you wanna set.

Here, I'll set a duration of 300 milliseconds again,
but you can, of course, play around with that value.

Now, with that, that's a good start,
but you will see that if you save this and restart the app
and you then go to such a meal,
nothing will change here.

There is no animation of this icon.

Because even though we set a duration
and the widget that should be animated,
we haven't given animated switcher any information
regarding which animation it should play.

What should it do with that child?

Well, that's the part which we'll add now.

Animated switcher also takes a so-called transition builder.

And transition builder wants a function
that receives a widget and an animation.

So therefore here we can add such a function.

And as mentioned here, we get a widget and an animation.

So I'll name the widget child
and the animation, animation.

Though since this here is your function,
these parameter names are also up to you.

And these parameter values here will be provided by Flutter.

That's really important.

You don't need to create an animation manually here.

Instead, the animated switcher will do this for you
behind the scenes.

It will create the animation and start the animation,
and it will start the animation whenever the child changes.

So it will automatically detect this
and play the animation as the child updates.

And that's why it's an implicit animation.

You don't need to worry about creating, or starting,
or stopping, or configuring the animation.

That will all be done for you.

That being said, though, this function,
which you pass to transition builder
also must return something which must be defined by you.

It must return a widget,
but it shouldn't return any kind of widget.

But instead, it should return
one of these transition widgets,
which I already mentioned earlier in this section,
like fade transition or size transition
because these widgets, these transition widgets,
then describe how you want to animate.

So with animated switcher,
we say that we want to animate
whenever the child widget updates or changes.

And in the transition builder,
we now return how we want to animate.
And here I'll return a rotation transition widget,
which will rotate this child widget whenever it changes.

So it will give it a spin.

Now, for that, it wants a turns value.

And turns is not a number like three,
but instead an animation.
Now, thankfully here we get an animation automatically,
so we can use this animation here.
With that, we're almost done.
But rotation transition also takes a child.
And that in turn wants a widget,
the widget that will receive this rotation animation.
And that would be this child here,
which in the end is this child.
And yeah, I know that's a lot of children,
but in the end, it's this child here,
which we set on animated switcher,
which will control when this animation switcher activates
and plays its transition.
It's also this child,
which is provided by animated switcher
to this transition builder through this first argument here.
So this child here is whatever you set as a child here.
And it's now this child,
which I wanna set as a child for rotation transition
so that the rotation animation is played on this child,
so on this child in the end.
So that's how all these children here are working together.

7. Configuring Implicit Animations

And with that finally,
we got all the setup we need to animate this icon.
And of course here I still took some time to explain this,
so it took us some time to build this.
But of course that's the case
because you're currently learning it.

If you take a look at this code, it should be easy to see that this is much less code than you had to write with the explicit animation.

And that's why implicit animations often are the better choice.

Now with that, let's save it and let's see whether it works though.

If I go back here to a meal and I click this star, huh, still nothing happens here.

And the reason for that is that at the moment with this code here,

Flutter can't really detect whether anything changed because sure, I am swapping the icon data inside of my icon, but it's technically still a icon widget and therefore an icon element that is visible on the screen and Flutter therefore doesn't see any difference.

It doesn't explore the detailed data stored in this icon, it just see that it was an icon widget before and that it's now still an icon widget.

So to make sure that Flutter is aware of this change, we should add a key to our icon here.

And you learned about keys earlier in the course.

They are always useful if you need to differentiate between two widgets that are of the same type but got different data attached to them.

And here we must add a key

because the AnimatedSwitcher will take that key into account

to find out if we technically have a different widget than before, even though the type might be the same.

And the key I will add here is a value key,

which is `isFavorite`

and therefore it will indeed change between true and false.

And Flutter will therefore see that something changed here and will then trigger this animation.

So now with the key added here, if we restart this and go here, now we see an animation.

And of course here we got a pretty wild animation because we're using the default animation between zero and one as a turns value here.

So we're in the end giving this star a full turn here, which looks rather wild.

To change this,

if we're not happy with this kind of animation, we could replace the animation we pass as a value to `turns` with a `tween` about which you also learned before, and which you can also use in the context of implicit animations

because the whole idea behind a `tween` really just is that it gives you more control over between which values you wanna animate.

And here we could then, for example, create a `tween` where we wanna begin at 0.5 and end at one.

And then again, Flutter would take care of translating its animation between zero and one to an animation between 0.5 and one.

You now just have to call `animate` here because otherwise you just got a `tween`.

And what you actually need is an animation and `animate` returns such an animation.

Now, `animate` then needs a parent and previously that was the animation controller, but in the end, an animation controller

is just a animation configuration object.

And the same is true for this animation here.

So you can simply pass this animation here

as a value to animate.

Now, the only remaining problem here is that by default,

Dart infers this to be an animation of type num.

So a mixed set of numbers, doubles and integers mixed.

And one way of working around this is to set this

to a double by using 1.0 instead of one

because otherwise this is inferred as an integer.

An alternative would be to take advantage of the fact

that tween is actually a generic type

and by explicitly telling it

that the values should be treated as doubles.

So either approach works,

but with that, we now changed the values

between which Flutter will play an animation.

And if we do that, we will therefore see

that now this is a bit less wild,

and of course you can still change this a bit more,

and, for example, animate between 0.8 and 1

to reduce this effect even more.

Like this.

It's really up to you what you wanna animate here.

But with that, I think this looks quite nice,

and this is how you can add an implicit animation

simply by using AnimatedSwitcher

or one of the many other animated widgets

provided by Flutter.

The difference between those widgets and AnimatedSwitcher

in the end mostly is that AnimatedSwitcher only cares

about the fact that you're transitioning between two values

and then you get full control over how to transition
whereas most of these other widgets are already focused
on one specific kind of animation that will be played.
For example, `AnimatedCrossFade`, which will simply fade
between two different widgets instead of allowing you
to decide which transition should be played.
As always, it is simply best if you do dive
into the documentation here as well,
and you simply play around with these different widgets.

8. Adding Multi-Screen Transitions

Now to conclude this section,
there's one last animation which I want to perform,
and that's an animation across multiple screens.
When I select a meal here on this meals list screen,
and we then transition to this meal detail screen,
I'd like to animate this image
so that it slides from this screen to the next screen.
And whilst this might sound super fancy and complex,
it actually, thankfully, isn't
because Flutter has built in support
for animating widgets across multiple screens.
And for that here, we can go to the `meals.dart` file
where I have this list of meals,
and then actually to the meal item,
because that is where I do have the image of my meal.
Here.
And we now have to wrap this image,
so in this case, this `FadeInImage` widget,
which is responsible for outputting the final image,
with another widget.
And that other widget here is the so-called Hero widget,

another widget built into Flutter.

And whilst the name might be strange,

this simply is a widget that exists

to animate widgets across different widgets,

across different screens typically.

For that, Hero needs a child,

which is the widget that should be animated.

But in addition, it also needs a tag,

which behind the scenes will be used

for identifying a widget on this screen

and on the target screen.

And here, I'll give this a tag of meal.id.

It should be a tag that is unique per widget.

And of course here, I have multiple meals

and multiple images.

So therefore every image should get its own tag

when wrapped in Hero.

And the ID is unique per meal

and therefore is the perfect tag per image.

So that's the first step.

We wrap the widget that should be animated,

and you are not limited to images, by the way,

though images are a common case.

But you wrap that widget that should be animated with Hero

and give it a tag.

And now you have to go to the place

where you want to animate to.

And of course, there must be a logical connection

between these two places.

For example, here we're on the screen

where we're coming from,

and now I'm going to the screen

where we are going to, the meal details screen.

And here in this meal details screen,

you have to find the same widget

to which you want to animate.

So in my case here, this image.

And it does not matter that this is now not a fade in image,

but a network image.

The only important thing is

that it's generally the same content,

and that will be the case here.

And now you wrap this widget here with Hero as well.

So I'm wrapping my Image.network with Hero,

and you have to give it exactly the same tag,

so again, meal.id here.

Because that's important for Flutter

to identify the widget on both the start

and the target screen.

That's all you have to do.

If you now save this and restart,

you will see that if you go to this list of meals

and you tap a meal, this image now animates

between the screens, forth and back.

And that of course, makes this animation look a bit nicer

than it did before.

And as you saw, adding this is really simple.

9. Module Summary

And that's it for this course section.

In this course section,

you learned how to add animations.

And you learned that there are explicit

and implicit animations.

And you did learn how to set up your own explicit animations by creating and initializing such an Animation Controller, which as the name suggests, configures and controls an animation, which you then have to start manually, and which you then can use in your widget tree to animate whatever you wanna animate with help of Animated Builder.

Here, for example, we animated some padding, or later, a slide transition where we even used some animation data, the animation value to be precise, and translated it to different start and to end values with help of Tween.

We even fine tuned the animation duration distribution by adding such a curve configuration.

That's how you can add an explicit animation.

And as you can see, this of course, gives you a lot of control and power, but it also increases the complexity, of course.

And as you also learned, often it might not be needed because here in the meal details screen, we then used Animated Switcher to switch between two different widgets with help of a key being set on the widgets.

And we played a rotation transition by just adding this code here, which is of course, less code than for the explicit animation.

Last but not least, you also learned about this Hero widget, which can be used to easily animate a widget across different screens.

Here from meal item, which is part of the meals screen
to meal details, where we also added such a Hero widget.
And that's how you can add animations to Flutter apps.
As mentioned, definitely also take a look
at the official documentation
to learn more about all these different
pre-built implicit animation widgets,
and possibly also to learn more about animations in general
if you want to.
But with this section, you now got the basics down
and you now know how you can animate widgets in your apps.

10_shopping-list_form-n-handling-user-input

1. Module Introduction

Now, in this course, up to this point,
you already learned a lot about building user interfaces,
and also about adding interactivity to your user interfaces.
You, for example, also learned how you can handle user input
and generate data like an expense item based on that input.
Now, in this course section, we're going
to take another closer look at handling user input
because in this course section you will learn
how you can build and use forms in your Flutter apps.
And you will of course therefore also learn
what such forms are and why you might want to use them.
As you will learn in this course section,
forms can, for example, help you with validating user input
and showing on-screen validation errors.
In addition, you're also going to learn how to get the data
that was entered by the user, how to submit forms,
and how to possibly reset them,
so that by the end of this course section
you will not just have built yet another Flutter app,
which of course also is nice and a great practice,
but you will also have more tools
when it comes to handling user input.

2. Setup & A Challenge For You

Now before we take another look
at handling user input,
and before we dive into this form feature
I mentioned before, let's actually start this course section

with a challenge for you, with a chance for you
to practice some of the things you learned up to this point.
I want you to build this app here, this shopping list app,
which at the moment doesn't yet allow users to add new items
to the list
because that is what we'll work on together thereafter,
but which at least shows some dummy grocery items in a list
that looks like this.

And my challenge for you is to build this user interface,
build such a list, add some widgets as needed,
try to come as close to this UI as possible.

Also, build and use any data models you might need.

And once you're done with that,
definitely take a look at my solution,
which you'll see in the next lecture
before we then thereafter dive into handling user input.

Now, one very important note, attached to this lecture,
you find a main.dart file which you should use instead
of your main.dart file where I do set up a basic flutter app
with a very basic theme, which you, of course,
could also adjust to your personal preferences.

But this is the theme we will use in this course here
or in this section.

And attached, you also find a data folder,
which you should unzip and move into your lib folder.

And in there, you'll find a categories.dart file
with the available categories

that we'll use in this app here for our grocery items.

And you'll find a dummy items.dart file in this data folder
where you find some dummy grocery items,
which you can use for a start to display them on the screen.

Now if you take a look at those files,

you also see that I'm importing some models in those file,
a category.dart file,
and a grocery item.dart file for example.

And you might also see if you take a closer look,
that I seem to be using a enum here as a key
for my different categories.

So you might also want to add a enum
that supports these values.

And you also definitely wanna set up a category
and a grocery item model in a to be added models folder
in files named as you can see them here in the imports.

And create those models so that you can set up
these two files here,
such that they use the models without errors.

Now attached to the next lecture,
you'll find my finished model files,
and you could also use those if you struggle
with setting up those models.

But definitely take it as a practice opportunity
and try adding those models on your own.

Either way, definitely try building this UI.

And then in the next lecture,
we'll build the models and the UI together.

3. Challenge Solution 1 - Building & Using Models

So, did you succeed?

Let's now build this together.

And for that, I'll actually start with the models
because we'll need those models to make the data,
the dummy data and the categories work.

And thereafter, I'll work on the UI.

So for that, I'll add a new models folder

to my lib folder here.

And in that models folder, I will start

with a new category.dart file,

which in the end will be used by categories.dart

because here I'm trying to import from it.

And there I wanna import this enum

and a category class, which can be instantiated like this.

So which takes a category description as a string

or a category title as a string as it seems, and a color.

So in category.dart, I'll start

by setting up this enum Categories.

The name is up to you in general,

but here, I'm referring to a categories enum.

So hence the name I chose,

which the end simply should contain all these values

you see here.

So vegetables, fruit, meat, dairy, and so on.

So here, I'll grab this and add my vegetables, the fruit,
meat, dairy.

And then I also have carbs, sweets, spices.

So we'll add that.

Carbs, sweets, spices,

and then also convenience, hygiene, and other.

So convenience, hygiene, and other.

And these are simply the kind

of dummy categories we'll work with in this app.

Of course, you could add more for this app.

That's all I wanna use.

And with that added,

this error here should already be fixed.

But, of course, we still must add such a category model
to really make this file here work.

So therefore, I'll go to category.dart.

And besides adding the enum here, I'll add a class in here, which I'll name category.

And this class in the end should have two properties.

It should have a title property, which can be of type string because that makes sense as a title.

And I wanna store the color, which should be of type color.

Now the exact property names of course, are totally up to you,

but these are the property names I will work with.

Now the color type is not built into dart, it's coming from flutter instead.

So to make it available in this file, we should import from flutter the material.dart file into this category.dart file, because this then unlocks the color type.

Now of course, to instantiate this class here and create objects based on it and assign these values upon instantiation, we also should add a constructive function here.

And it can be a const constructor function like this.

And then here I simply wanna accept this.title and this.color.

And I'm using positional arguments here, so I have not wrapped them with curly braces because here where I create my category, I also assigned these arguments by position and not by name.

I don't have something like Title Colon here.

Instead, it's just the order that matters here for assigning these parameters.

I also still got an error here regarding my fruit enum value because I have a typo here.

Of course, we should delete that.

And with that, with the type of fix
and with this basic model set up here,
this categories.dart file works
and doesn't show any errors anymore.

Of course, you should make sure
that the import here is updated
and uses your project name here after package.

But with that, we got the first thing out of the way
and we added this category model, which is great.

And with that, we can already move on
to the grocery item model.

So for that, in the models folder,
I'll add grocery_item.dart as a file name.

And in there, I wanna set up
yet another class which then can be used
by importing it here in dummy_items.dart
and by then instantiating items like this.

So by using named parameters as you can see,
and these named parameters specifically, so
therefore, we can go to grocery_item.dart
and in there, also at a new class GroceryItem
because that's the class name I am using in dummy items.

And then there, as it seems, we will have an id parameter,
a name parameter, and a quantity parameter,
and a category parameter.

And therefore, probably also properties
with those same names.

An id and name are strings as you can see,
quantities, a number,
and category seems to be of type category.

So that category class, we just added the type

of that class.

That seems to be the type of
that category parameter and property.

So therefore here, when setting up this grocery item,

I wanna add my id, which will be of type string.

I will add my string name,

my quantity, and that should be a number.

And here we could have a double or an int.

And I'll go for an int here

because at least in this app here,

I don't wanna allow quantities like 1.5.

Instead, you either have one or two items.

So only full numbers.

So that's why I'm using an int.

And last but not least, I also will add a category,
which should be of type category as mentioned.

So here we can use that category,

which we must import from our package here,

from our project, from the models folder

and the category.dart file.

And with that added here, we can

and should add a constructor,

which can also be `const GroceryItem`.

And here, as you can see, I am using named parameters.

I have a name and then a colon,

and then the parameter value.

So therefore here, I'll use curly braces

and then assign my values to id, name,

quantity, and category.

Now all these should also be required here

because these properties must not be null,

therefore, we must require these parameters.

And as you learned, named parameters
by default are optional.
So to make sure that they are provided, we can
and should add required here in front of every parameter.
And that's then the grocery item model finished.
Now with that, we're almost done,
but I'm still getting an error here in dummy items.
And in the end here, we learned
that we should try removing the const keyword here
because we're dynamically accessing the categories here.
And of course, to make sure that categories works here,
that this enum is available here.
We also should definitely import from the category.dart file
in the models folder
because that's where this enum is defined.
So after finding
and fixing those extra errors, you also finished this part,
you added the models now and you're using them correctly.
And you also find my finished files
attached to this lecture now.
The finished data folder with its files in the final form.
And, of course, the models folder with its files,
you find a link to all these files attached to this lecture
so that you can compare your solution to mine.
And you can also use my files in case you got stuck
or you were not able to reproduce this.
And with that out of the way, we can now move on
and start working on that UI.
Before we then thereafter dive into the actual core content
of this section and take a closer look at flutter forms.

4. Challenge Solution 2 - Building the List UI

So let's get started with that UI then.

And for that, in that lib folder,

I'll add a new folder called widgets.

And you don't have to do that of course,

but I will add it here to keep my files,

my source code files, organized,

and make it clear that everything in there will be a widget.

Whereas in models I have my model classes and so on.

And in that widgets folder,

I'll therefore add a new file

called grocery underscore list dot dart,

because this app is about outputting some grocery items.

Now of course it's very possible

that you chose different file names

and also different widget names

or a different widget structure.

And that's of course all fine

because ultimately this was meant as a challenge

and as an exercise opportunity for you.

But here I'll work with that widgets folder

and the grocery list dot dart file.

And in that grocery list dot dart file,

I now want to add a new widget class

and I'll name it grocery list

because well that's what this widget will be about.

And to make it a widget class,

we of course have to extend something, as you learned.

And here I'll extend a stateless widget

because whilst we will actually change this later,

at the moment for just outputting these dummy items,

we don't need a state full widget here,

instead a stateless widget is all we need to use.

And we should import this from Material Dart.

And with that added we can and should add our build method
and of course also this constructor.

And here I'll add my grocery list constructor like this

and only accept and forward that key

because we don't need anything else here in this widget.

And now with that done here in the build method,

I of course wanna build that list user interface.

Now for this, I'll first of all start with a scaffold widget

to make sure that we have this basic screen set up

because otherwise we would have this black background

and would have almost no styling.

So I add my scaffold here

and in there I'll actually set up an app bar

with the built-in app bar widget,

as we did it many times before in this course.

Simply so that in there I can set my title

to "Your Groceries", like this.

And of course title should be a text widget

instead of a string.

So we set it up like this

and can turn this into a const widget.

So that's the app bar.

Now for the body, I wanna output that list

and this list could potentially become very long.

Now of course not with the dummy groceries I have here.

There, I only have three items,

so if we only ever had those three items,

it would of course be absolutely fine to output that list

with the column widget, for example.

Or with the list view without using its builder mode.

But the idea with this app here, of course,
is that ultimately we will be able
to add more grocery items to it
and therefore this list could become very long
and we could have dozens
or hundreds of items on that list potentially.
And therefore you learn that it is preferable
to build such lists with list view
and there with its builder constructor function.
Because this will then build the list in an optimized way
and make sure that we only display
and render the items that are actually visible
or about to be visible.
So here I'm using this list view builder widget.
And with that added here,
we then have to add our builder function here,
which is a function that takes some context as an input
and then the index off the list item
that is currently being built.
And these argument values
will be passed in automatically by flutter, as you learned.
Here we then just have to return the widget
that should be used for every row.
And here you could, for example, use the row widget
or anything like that.
But I will actually go for the list tile widget here.
Since this comes with some built in list optimized styling,
which makes our life a bit easier
when it comes to building this list here.
So I add this list tile here,
and now with that added, in that list tile widget,
I, for example, wanna set the title.

And the title should be set to the title of the grocery item
or to the name of the grocery item, to be precise.
So therefore to output that name here as a title,
I have to get access to my different grocery items,
so to the dummy items at the moment.
Well and to do that we should simply add an import
to this dummy items dot dart file in this file here,
by importing our own package,
which in my case is called shopping list.
And then from there the data folder,
and then there the dummy items dot dart file.
And with this import added here,
we can then go down here and use our grocery items list
to access an item by index
and to then access its different properties, like name.
Though of course, title actually does not want a string
but a widget.
So here I'll build a text widget with that name,
but that's now my title.
Of course we also have to make sure
that flutter really goes through all our grocery items
and that's why besides adding item builder
to this builder constructor function,
we should also add item count.
And this should then be set to grocery items dot length,
like this, so that flutter knows how often
it has to call this builder method potentially.
Now besides setting the title,
I also want to add the leading parameter here
to this list tile to output a little indicator
for the category this grocery item belongs to.
And here it's important to keep in mind

that all these categories have colors attached to them.
So in the end it's this color which I wanna output here.
And for outputting such a colored block,
which in the end is the goal here,
you have different options
and I'll simply go for the container widget here
because this container widget allows me to set a width.
In this case I'll go for 24 pixels
and a height, also 24,
and then a background color of this
widget block I am rendering here.
And here I'm simply rendering a square with a certain color.
The color here of course will also be grocery items index
to reach out to the item we're currently rendering,
dot category to get its category,
and then every category object has this color property.
So that's what we need to use here now on this category,
we access color to set the color of this block.
Well and last but not least here,
I'll add the trailing parameter to this list tile
because there I want to output the quantity
with the text widget and then again,
by reaching out to the grocery item by index
and then there we have quantity.
Now text, as you learned, of course,
wants a string, not an integer.
And we can easily convert this to a string
by calling to string here.
And that's now my finished list with its list items.
Of course, we could also outsource this list tile here
into a separate standalone widget.
Here I'll keep it all in one widget though.

But with that added,
we should be able to now use this grocery list
as a widget here in the main dot dart file
as a value for home.
So that's what I'll do here, grocery list.
And we also must add a import here
to package shopping list widgets grocery list dot dart.
We can then also add const down there.
And with all that, we can run this app here
and we should then be able to see
this basic grocery list UI on our emulator.
Here we go, that's looking good to me.
And with that added,
we can next work on this new item feature,
which allows us to add new items to the list
and we will of course add that feature
by using this flutter forms feature I mentioned earlier.
As a side note, you find a link to my finished project
at this point of time attached to this lecture
so that you can, for example,
compare your grocery list widget to mine
and also simply copy and use my code
so that you can follow along with exactly the same code
as I have it here at this point in the section.

5. Adding a "New Item" Screen

So, let's now dive into this forms feature
and learn how it can simplify the process
of getting and handling user input.
Now for that I'll add a new widget,
which I'll name new_item.dart
because we'll have a separate screen for adding new items.

And in this new_item.dart file,
I'll therefore add a new class, a new widget class,
which I'll name NewItem,
and which extends StatefulWidget here
because we'll use some StatefulWidget features
in this widget here.
So therefore, this should be Stateful
because we'll need to manage some state later.
And therefore of course, we also have
the NewItemState class here, which extends State,
a State which is connected to the NewItem widget though,
like this.
And then here, in NewItem as always, we have createState,
where the State also refers to NewItem
and where we then return a new instance
of our NewItem state here, like this.
And of course, we also should add
the constructor function here,
which can be a const constructor function,
NewItem, where I only accept and forward this key.
So really, a basic constructor function.
And in the State we then have our build method
of course, as always.
And in here, we will then build
this NewItem screen in the end.
So, this form that should be displayed.
Now, I will indeed load this as a separate screen,
and therefore we could have also annotated this widget
with Screen here at the end.
But since in this basic app,
I will only have these two widgets and both will be screens,
I will not make that differentiation here.

But you could, of course.

But here, I'll have this widget which will be a screen,
and therefore I will of course start by returning
a scaffold widget to have that base screen set up,
as always.

Now in there, I then want to have my AppBar
to set a title in this place.

And this title here is simply some const Text,
a const Text widget, where we could say Add a new item,
because that's what we're doing on this screen.

We're about to add a new item to our grocery list.

Of course, then in Scaffold we also have our body,
and this body will in the end be this form
on which we're working.

Now before adding that actual form,
I'll add the Padding widget though, to add some padding
to the left and right and top and bottom.

So, some spacing between the form inputs
and the edges of the device screen.

And for that, we have to set the padding parameter here
on the Padding widget.

And I'll send it to a value of EdgeInsets.all of 12.

The value of course, as always, is up to you.

And we should add const here,
because this is a const widget.

At the moment, I could also add const here
and remove it here in front of EdgeInsets,
but this will change later on.

So, therefore all only add const here,
and not add it in front of Padding.

And with that, as a next step on the Padding widget,
we have to set the child parameter.

And this will now hold the actual form

that we're going to add.

Now we're going to get started with that form

in the next lecture.

For the moment, I'll just add a dummy Text placeholder,

The form, which we will get rid of in the next lecture.

But I will add it here, so that we can simply see

whether we can reach this screen

by going back to `grocery_list`.

And there, I now wanna make sure that we have a button

that takes us to the `NewItem` screen.

And that button here should be in the `AppBar`, because there,

besides having that title, I want to have some actions,

or one action, in this case here to be precise.

And that one action should be an `IconButton`,

which we can tap to go to this `NewItem` screen.

And `IconButton` of course, wants the `onPressed` parameter,

as you saw before in this course, and the `icon` parameter.

Now for the icon, I'll render a new icon

with the built-in `Icon` widget.

And I'll use the `add` `Icon` here.

And for `onPressed`, we want to add a function here

that takes us to the new screen.

And we could do this all here in an anonymous function,

and that would be all right.

But here, I'll actually add a new method,

`addItem` could be a fitting name.

And this method doesn't return anything at the moment,

therefore I'll add it like this.

And it's now `addItem`,

which I want to assign as a value to `onPressed`.

So, this `_addItem` function I just added here.

And inside of `addItem`, I will now use the `Navigator`,
which is built into `Flutter`,
about which you learn before in the course,
`of(context)` as you learned.

And context here is not available at the moment
because we're in a `StatelessWidget`,
which unlike the `State` class,
does not have a globally available context widget.

And therefore we could either accept it here as an input,
and then of course make sure that `addItem` gets called
to such that the context is passed along.

For example, by wrapping it here into an anonymous function
so that we have full control over how it will be called.

And then we could forward context like this.

We could do all of that, but since I'm saying could,
this is probably not what I'm going to do.

Instead, I'll go back to this version here
and not accept `BuildContext` as a parameter here.

And instead, I'll convert `StatelessWidget` here
to a `StatefulWidget`,
because we will later need to manage a list of items
in this widget here anyways, and therefore,
this will have to become a `StatefulWidget` later.

Hence, we can already make this conversion,
right-click on `StatelessWidget`,
and refactor and convert it to a `StatefulWidget`.

And with that conversion done, we can now use context here.

So with that back here in `addItem`,
I can use `Navigator.of(context)`.

And then make sure that we go to the `NewItem` widget
as a screen by using `push`,
as you learned it before in this course.

And now here, we want to push a `MaterialPageRoute` like this.

And this then takes the builder function

where we get this context for this route,

and we then simply return our `NewItem` widget here.

And for that, you just must make sure

that you also add a import to the `new_item.dart` file

to this `GroceryList.dart` file.

But then we have this added here,

and we can go to the `NewItem` screen.

And here, we can actually add `const`

because we get a `const` constructor in a `NewItem`,

and we don't pass any dynamic value

or a function as a value to `NewItem`.

Hence, we can use this `const` constructor like this.

So with that, we have the add item method,

and we should be able to go to this `NewItem` screen.

If we save everything, we now see that plus button,

this add button here in the `AppBar`.

And if we tap it, indeed,

we're taken to this `NewItem` screen.

And now we can work on this `NewItem` screen

and add the form to it.

6. The Form & TextFormField Widgets

So let's now go back to the new item

dot dart file and replace this text place

holder widget here with that form we want to add.

Now a form is simply a combination of input fields

and in flutter there is a built-in form widget,

which you can use

to build such a combination of input fields.

Now of course, we already handled user input

before in this course,
and we did that without using that form.
So clearly we don't need to use a form in flutter
to display multiple input elements and get some user input.
But using this form widget comes with some extra features
that can help with getting that user input, validating
that user input, showing onscreen validation
errors and so on.

That's why you might want to use it, especially
for more complex input field combinations.
So here we can create such a form by using the form widget,
and as you see, I'm getting an error
when adding it like this.

I'm getting an error that I must add the child parameter
and that must be added
because the child parameter ultimately will control
or will define which widgets are inside of the form.
And here you can use any widgets you already know.
For example, the column widget,
since I will have multiple rows of inputs in this form.

So we can still use the column widget
as we did it many times before in this course.
Therefore, here on column, we can set the children parameter
and this then takes our different rows in the end,
the different widgets
that should be displayed above each other.

And here I wanna start with an input field
that allow us the user to enter the name, the title
of a new grocery list item.

And before in this course, in the expense tracker app
for example, we did that by using the text field widget.
This is a widget we can use for displaying an input field

where users can type and enter some values.

Now, when using the form widget,
you should instead use the text form field widget.

So instead of text field, which we used
before, it should now be text form field
because this is a widget that integrates with
that form widget and this set of features
provided by form and
therefore it's text form field that should be used here.

And if you just type form here
and you open this suggestion menu, you will actually see
that as you scroll down, you find many other widgets
that are meant to be used inside
of a form like this dropdown button form field widget,
which would be the form alternative
to the dropdown button we learned about
earlier in this course.

So we have these form specific input widgets like text form
field, which are meant to be used inside of a flutter form
and which simply support the different features
provided by the form widget.

But with text form field added like this,
we can still configure it
and you can, for example,
still set the max length parameter,
which you saw earlier in the course on the text
field widget already.

We can set it to 50 to make sure that we don't allow titles
or names that are longer than 50 characters.

We can also still add the decoration parameter,
which we saw earlier in the course,
and set this to an input decoration object where we,

for example, add a label.

So this also does not change.

We can add a label of name here,

and this should be a const value.

And the label should actually be text widget here
of name in this case.

So these fields can still be added.

And if you open this suggestion menu

by pressing control space, you will see that many
of the parameters you see here were also available
on text field.

But there also are some special parameters available.

For example, the validator parameter.

The validator parameter allows you to add a function
to this text form field, which takes a string value
as an input, a value that will be produced
and passed in automatically by flutter,
and which then also returns a string value,
a value defined by you.

So here we can add a function,

an anonymous function in this case for example,
which automatically receives a value which will be
of type string or null,
and which then should return a string value.

And the idea behind adding this validator parameter
with this validator function is that you will be able
to tell flutter when this function should be executed
because your inside of a form,
and as you will see later, you will
therefore be able to tell flutter when to validate this form
and when flutter then calls this function,
you can return an error message if validation fails.

So if you determine
that an invalid value has been entered into this form field
and flutter would automatically display this error message
which you are returning here,
or which you might be returning here on the screen
after it ran validation.

And you'll see this again later at the moment,
I'll just return demo here.

And at the moment we actually won't call this validation at
any point of time, but I will get back to this later.

But that is one of the new parameters you can add now.

And I am mentioning this right now already
because it is to understand

that text form field is basically an extended version
of the regular text field
that provides extra features like validator,
which in the end are enabled by form.

So that's how you can add such a text form field.

And I will get back to validator later as mentioned.

But this is of course not all I want to add to this form.

So let's continue.

7. A Form-aware Dropdown Button

Besides adding such a text form field,

I also wanna add a row thereafter, a row which allows me
to display two other widgets
next to each other horizontally.

And the two widgets I wanna display here
are another TextFormField,

which can be used to enter the quantity of items
that should be added.

And next to that though, we will

of course, configure this in a second,
but next to that, I wanna have a dropdown button,
which can be used for choosing a category.
And here, we don't wanna use the regular dropdown button,
but instead, we wanna use
the `DropDownButtonFormField` here, which again,
is the form-specific version
of this dropdown button widget
that adds extra form-specific features
or supports those form-specific features.
Now, besides adding some new features, it still works
as the old dropdown button though, as you can already see,
it wants some items here
and an `onChanged` method, which will be triggered
whenever a new item was picked.
But still, this is
how you can add this dropdown button form field.
And I'll also add one more widget in this row
between these two widgets.
I'll add a `SizedBox` for some extra spacing.
So here, I'll then add a width of eight.
Now for the configuration of these widgets, I'll start
with the `TextFormField`.
And here, I'll set the decoration parameter
to an `InputDecoration` object that allows me
to add the label to this widget here,
a text widget as a label where I say Quantity
because that is in the end the label I wanna add.
And that is my `InputDecoration` therefore,
that's all I wanna add.
And I will also set an initial value here,
which is another special parameter I can add

on this TextFormField, which is not available
on the regular text field, which allows me
to set an initial value
that will be displayed in this TextFormField.
And here we could, for example, set this to one as a value,
very important, not as a number, but as a string
because TextFormField,
like text field in the end works with strings.
Every user input is a string, even if it's a number,
but it is a number as a string then, and the initial value,
therefore, also must be a string, in this case,
the string one.

So that's the basic text form field configuration
I wanna use here.

Now for the DropdownButtonFormField I, of course,
wanna output my categories here as items.

And therefore, here, I'll add a list.

And in this list, I'll use this for loop this in list
for loop, which you saw a couple of times already.

And we can then use this for loop to go through
all the categories like this.

And for categories to be available here,
we must import categories from the categories.dart file,
with that import that we can use it,
but we don't use it entirely like this.

Instead categories, if you take a look at it,
is actually a map, not a list.

And we can't loop through a map like this.

Thankfully, it's quite easy to convert it to a list though
by using the special entries property
that is available on all dart maps.

So this is a property provided by Dart on every map

and entries gives you an Iterable,
which in the end contains all your map key value pairs
as items in that Iterable.

So in that list in the end, so here, we can loop
through all our category entries then,
and we get these entry item objects then.

And for every category in categories.entries,
we can then output a DropdownMenu item.

So this does not change.

There is no form specific version of this widget.

It's still a DropdownMenu item, which wants a child widget.

And the child here, in my case here will be a row
for every DropdownMenu item

because I wanna have two items next to each other here
and here in that row, I wanna have a container,
which again will be a box with the color of the category
that's being displayed here.

Then next to that, a SizedBox for some spacing here
with a width of six, for example.

Then a text widget.

Now in the text widget, I wanna output the title
of the category.

And this can be done by using this category here,
which we're getting from category entries.

And then there, we'll have a key and the value property,
because again, we're operating on those entry items yielded
by the entries method in this list of entries.

Now every entry item will have a key value
that in the end gives us the key we had for
that item in the map
and the value, which as you might guess, is the value
for that key in the map.

And here, I'm interested in the value
and that they are the title.

Now in my container here, I'll give this container a width
of 16 and the height of 16 to render such a square.

And then add the color,
which here will be `category.value.color`,
and these should all be children of this row.

So we should move all these widgets into the children list
of the row like this.

And that should then display fitting dropdown menu items.

Now, as you also might recall from earlier in the course,
you can and typically also should define which value
will be assigned to every `DropdownMenuItem`.

So the value that will then be available
in the `onChanged` function here later
whenever the user picks a different `DropdownMenuItem`.

And for that, you can add the value parameter
to the `DropdownMenuItem` widget.

And here my value should be
the overall category value object,
which is that object with the color and title property.

We could also use category key to use the enum value here,
but here, I'll simplify things a little bit
and use the overall value object as a value,
which is something you can do and which I'm doing here.

Now we also must add a function
and we'll add the actual function logic later.

For the moment, it will just be an empty anonymous function
that is called automatically by flutter every time the user
picks a different item
and which will automatically receive
that item's value as an input.

So in this case, the category that was selected,
and that's the finished DropdownButton
in its form-specific variant.

If you re-render everything,
you should be able to see this.

But here, we actually don't, we don't see anything here
because if you take a look at the debug console,
we got an error and in the end, it's a rendering error
because we're doing something
which you already learned earlier, that it won't work.

We are in the end using a TextField inside of a row.
And I mentioned earlier in the course that TextField
in the end also is unconstrained horizontally as is the row.

And that's why we're getting an error here.

To work around this, you should wrap TextFormField
with the expanded widget like this.

And you should also do this for the dropdown button
because for that widgets, it's actually the same.

So here, we should also wrap expanded
around the DropdownButtonFormField widget like this.

And if you do that, if you do add expanded
around these Q widgets and you then save everything
and restart, you should be able to go there
and see those widgets.

Now to make sure that they are also aligned correctly,
I'll then also go to my row
and they are at the crossAxisAlignment and here,
set this to crossAxisAlignment.end.

Now with that, we can see that this is aligned properly
and you will also see that indeed, if you do choose
value in the dropdown, it is displayed here.

And you might recall that for the expense tracker

with the regular dropdown button, this was not the case.
So with this dropdown button in its form-specific version,
indeed, we now do have that extra-functionality
of our selected item being stored.
And that is another feature provided by that form.
But of course, we still had no item initially,
but that is something we'll take care of later.
We do have these basic inputs though,
and with all these inputs added, we can next add the buttons
before we then thereafter take a closer look
at those forms-specific features
and functionalities that we may now use.

8. Adding Buttons to a Form

Okay, so let's now finish this form
or this UI on this new item screen here
by adding one additional row here,
still in that list that belongs to the column,
but after this row with the dropdown button
and the quantity field.
And in this row here, I wanna output two buttons,
so two children, which I have here.
And the first button will be a text button
that should allow us to reset the form.
And the second button
thereafter should be an elevated button,
which we'll use to submit the form.
Now the text button here, of course,
also will need `onPressed` parameter,
which at the moment is set to an empty function.
This will change later though.
And the child here should be a text widget

where we can simply say reset
because that is simply the content of this button.
For the second button,
I'll also have an empty anonymous function right now.
And here we'll later make sure that we do extract the data
entered by the user and that we do submit it.
And then thereafter here for this child,
we'll also have a text widget, which will also be constant,
where we can say add item, for example.
So these are the two buttons that should be added here,
and with them added here, if we save everything
and restart this, we can see those buttons here.
And to make the styling a little bit better here
and make this look a bit nicer,
I'll actually go to this row, which contains the button
and set the main axis alignment to `MainAxisAlignment.end`
to make sure that the content of this row
is pushed all the way to the end
of the main axis of the row, which is the horizontal axis.
So that pushes the buttons to the right,
and I wanna have some spacing between this row
and the row before it.
And that can, of course, for example, be done
by adding such a sized box,
maybe here with the height of 12, so that this makes sure
that this UI looks like this,
which to me looks nicer than what we had before.
But with that, we now get these buttons.
The next step, of course, is to make sure
that they also do something when we press them.
For example, they should validate the user input.

9. Adding Validation Logic

So let's start

with validating user input.

For that, of course,

we already got this validator function here.

It's not doing anything meaningful right now,

but I mentioned earlier that this is a special parameter

that can be set on that text form field,

which in the end wants a function that will be triggered

by Flutter by this form in the end, as you will soon see,

to run some validation logic

and return an error message if validation fails.

Otherwise, this function should return null.

So if it does return null, Flutter knows

that validation succeeded and the entered value is correct.

Otherwise, if we return some text here,

this will be the error message

and this will tell Flutter that validation failed

and the entered value is invalid.

So therefore, here in this validator function, we can work

with that value which will be

provided by Flutter when this function here is executed.

And this value here will be the value

that's currently entered into this field,

and we can then use a if check to check whether it's valid.

For example, we can check if it's null.

This is not the only check I want to perform though.

Instead I will add another check here combined

with the first check through the logical or operator

so that even if this here is not true.

So if the value is not null,

we still perform the second condition.

And my second condition here is whether value is empty,
so not null, but an empty string.

With that, we would make it into this if block,
if either the value is null or if value is empty.

It's also worth knowing that if value is null,
this second check won't even be performed by Dart
because it already has one true value
if value is null,
and that is all it needs.

If at least one of the conditions is true,
we make it into the if block.

So in that case, the other conditions
wouldn't even be evaluated.

Of course, that's different when using the and operator.

Here, all conditions must be met
and therefore all conditions will be evaluated.

With or, one matching condition is enough
to skip the other conditions and make it into this if block.

But even with those two conditions here,
I don't have all the checks I wanna have.

I'll add another or operator
to add another condition, another check here.

And you can add and combine as many conditions as you want.

You can also combine or and and if you needed to,
but here I'll stick to or

because I have a lot of alternative checks which I want
to perform, and if one of them is true,

I wanna make it into this if block
because my next check here is whether a value
once trimmed.

And you might recall that trim is a built-in method which
you can execute on strings

to remove excess white space at the beginning and end.

So once the value is trimmed

and we got rid of any excess white space,

I wanna check if the length is equal to one.

Because here, for example,

I want to have input values, names,

that are at least two characters long.

So if the length is one, I'm not happy with that.

And actually we can check for smaller

or equal than one to make sure that if the value

after trimming it would have a length of zero.

We also make it into this if block.

And now there's one last condition I want to add here,

and that is that value

after trimmed has a length greater than 50

because I want to have a max length of 50,

which is also why I set this parameter.

And indeed setting this parameter should ensure

that you can't enter more than 50 characters.

But still it doesn't hurt to also add this check here.

And now I add the curly braces

to define my if statement body

and will now only make it into this body here if at least

one of these conditions is met.

Now, it's then in this body

where I wanna return my error message.

And of course here I want to say something more meaningful.

Like for example, must be between one

and 50 characters long or just characters.

And otherwise after this if statement, I wanna return null.

So this will execute if we don't make it into

this if statement, because then we do indeed have

a valid input value.

And then I wanna return null to tell Flutter
that value is valid.

If we make it into this if block here,
we'll instead return this, this error message
and that will tell Flutter that something went wrong.

Now with that, if you save that
and force a reload, you will see
that this has no effect though.

If I enter something here and I delete it and this is empty
and I for example, click add item,
I don't see this error message anywhere.

Instead we'll have to trigger this validation function here.

We'll have to tell Flutter to trigger it to be precise
and we'll do so in the next lecture.

But before we go there, I want
to copy this validator parameter
and also add that to my other text form field,
which is about entering a quantity.

There I will add it as well,
but here the checks will be different.

Here I wanna make sure that it's not null
or if it is null, I wanna make it into this if statement
and return an error message.

And I want to check whether it's empty,
but thereafter I'm not interested in the length.

But instead here in the end, I want
to get a number.

As a string, as you learned, it's always a string,
but I want to treat it as a number
and handle it as a number.

And to do that I'll add `int.tryParse`

so that I try converting the entered value to a number.

And if this is null,

I also wanna make it into this if check,

because `tryParse` returns null

if it fails to convert the string to a number.

As you can, for example, see it here,

this would not be a valid number,

hence we get back null.

So if we get back null, we know it's not a valid number.

And in that case I wanna show an error.

Alternatively, I also have a fourth condition

where I wanna parse this as well.

And here I will know that parsing in general succeeded

because if it would've yielded null, Dart would've stopped

after checking this condition as mentioned earlier.

So if this condition is evaluated, I know

that it won't yield null,

but here I wanna make sure that it's not equal

or smaller than zero

because the number which I get here

must be a positive number.

Now here Dart actually doesn't understand

that this will never be null here because of this check.

And hence we have to add an exclamation mark here

to tell Dart that once this check is executed,

this value won't possibly yield null because

otherwise we would've stopped after this check.

Well, and that's all my checks here.

If any of those conditions is met,

I will return an error message, which here could be,

must be a valid, positive number,

and that's the validation logic for this text form field.

Now, for the dropdown button, we could add a validator,
but we actually won't need one
because whilst we don't have an `initialValue` here yet,
we will soon add one
and thereafter it simply won't be possible
to pick an invalid value
because only valid values are suggested
and you need to pick one.
So therefore here we don't really need a validator.
We'll just have to make sure
that later we do set an `initialValue`,
which we can't do with `initialValue`
because this parameter doesn't exist here
for dropdown button form field.
But I'll get back to this later.
Therefore, this is all the validation logic needed,
and now it's time to make sure
that this logic also gets executed.

10. Getting Form Access via a Global Key

So how do we make sure
that this validation logic does get executed?
Well for that we need to tell Flutter
that for this form it should execute all the validators
that are registered for all the form fields.
And doing so is thankfully quite straightforward
because you can get access to this form widget
and the underlying object that's managed by Flutter.
And you can tell Flutter to validate this form.
Now you do get this access to form through a key,
and that might be strange
because we have used keys before in this course,

but only to uniquely identify,
which it's typically when used in a list.
Now here it's the same key parameter,
but it'll be a different kind of key that we add here.
It'll be a key that we should create as a property here.
And I'll name it form key,
and it will be of type global key.
So here I create a new global key
by using the global key constructor function,
the global key class which is built into Flutter.
And global key creates a, guess what, global key object
that can be used as a value for a key as we need it here.
Now the difference between a global key
and for example a value key
which we used before in this course is
that a global key here then also gives us easy access
to the underlying widget to which it is connected.
And it ensures that if that build method here
is executed again
because we set some state, for example,
this form widget is not rebuilt
and instead keeps its internal state,
which is very important
because it's that internal state
with which we'll work in the end.
It's for example that internal state
that will tell Flutter
whether to show some validation errors on the screen or not.
That's why such a global key should be created.
And in general when working with Flutter,
you will almost always only need a global key
for working with forms

because you can then also use this global key
to get access to that form.

And that is indeed what we'll do next.

Before doing that though,

I'll add a little state annotation here

because global key is actually a generic type,

and I wanna tell Flutter that the widget or the object

to which this global key in the end will be connected

will be a form state object.

That's in the end what this form widget gives us

under the hood.

And this will simply give us some extra type checking

and auto completion suggestions.

So we wanna add this annotation here.

With that added, I'll add a new method.

SaveItem sounds like a fitting name.

And it's this method that should be triggered

once this last button here is pressed.

So here, instead of pointing at an anonymous function,

I'll point at saveItem like this, not executing it,

just pointing at it,

so that this function gets executed

when this button is pressed,

and it's now in this function

where I wanna trigger validation.

And doing so is super straightforward

because with that global key added

and connected to the form, you can use it

to access the current state property

on that form key object

and tell Dart that this will never be null.

So in the end you are telling Dart

that your form key has been connected to the form.

And here we know that save item can't execute before this build method has been called.

And therefore this form has been created and the form key attached because the only way of executing saveItem is by pressing this button.

That's the only place where we mention saveItem.

And at the point of time this button can be pressed, the form will of course have been created, and the form key will of course have been attached to the form.

So therefore in here we know with certainty that this underlying form state object will exist.

And that's exactly what we're telling Dart here with that exclamation mark, that this underlying state object will exist and is not null.

And then on this state object, we can access various utility methods like for example, validate, which is a method provided by the form widget in the end, which will automatically behind the scenes reach out to all the form field widgets inside of the form like text form field and execute its validator functions.

That's what validate will do.

In addition, validate will return a Boolean value, which will be true if all validator functions passed or false, if at least one validator function failed, and this will become important later.

With this added though, we can save everything, reload and go back.

And if I now click add item, I get an error here
for the name because it's empty.
If I enter an invalid quantity like minus one,
I also get an error there.
By the way, speaking of the quantity,
I just see that we should change the keyboard type here.
So I will quickly do that.
Go to the quantity and set the keyboard type here
to text input type number.
This is of course not related to validation,
but something we should do
but detached from that, we now also get working validation.
If I do submit valid values,
those validation errors of course go away.
And that's how we can leverage this form widget
combined with this form global key here
to automatically trigger all these validation functions
which might return error messages,
which are displayed automatically by Flutter.
And that of course makes form validation much easier
than if you have to handle this manually.
So we learned how we can use the form key
to validate the input.
We'll soon also use it to save the input
and get access to the entered data.
But before we do that, we can also use this form key
and the form state to which we get access
through that key to reset the form.
And that's what I wanna do here on this first button.
We could also create a standalone method here
or do it in this anonymous function here,
which is what I'll do.

And in there we can also use this form key again
and its current state, which we know won't be null
to then call the built-in reset method,
which is provided by Flutter on this form state object
to which this form key in the end is connected
and this will reset all input values.

So with that, if we do that and we reload again,
if I enter some values here and I then click reset,
you see these input fields are set back
to their initial values.

And this can of course also be a super useful feature.

11. Extracting Entered Values

So now we can validate and also reset the form.
Of course, one crucial thing is missing, though,
and that's submitting or saving the form
and we can of course also do that.
Now to save the form
and get access to the entered values,
we can again use the formKey and there the currentState
and then call the save method.

Now when calling the save method,
a special function will be triggered
on all these form field widgets
inside the form like this TextFormField.

The special function that will be triggered
is the onSave function, so a function you pass
to the onSave parameter value.

Now onSave is a function that receives the value
that was entered as an argument
and this will be the value at the point
of time save is executed

and then in here you can do with

that value whatever you want.

For example, to get access to it and use it along

with all the other form values that have been entered,

you can add a new variable here

and that could be the enteredName,

which initially is an empty string, let's say,

and you can set this enteredName to a new value,

the currently set value inside

of this onSave function like this.

Now here we must add an exclamation mark

to make it clear that value will not be null.

Alternatively, you could also add an extra check

and for example, return if value should be null.

In that case, this code would not execute,

but this is redundant and not needed here

because we already have our validation logic

where we check if value is null.

Therefore we should only call

that save method if validation fails.

And to make sure that that's the case,

we can wrap this validate call here into an if statement.

So use it as a condition in a if statement

because as mentioned before, validate returns a boolean

and it returns true if validation succeeds

and false otherwise.

And then we can move this save method call here

into this if statement so that we only run this code

if validate returns true,

which as mentioned will only be the case

if validation succeeds.

Therefore, we'll then also know

that in all our `onSaved` functions like this one here,
the value will be valid because
otherwise we wouldn't call `save`
and this `onSaved` function here wouldn't get called.
That's why we know with certainty
that value here will not be null.
We also don't need to wrap this with `setState` here
because actually I don't need
the `build` method to execute again.
I don't have any UI update
that should be performed because of that.
I just wanna store the value in a property of my class here
because that then allows me to use that property here
after calling `save`, for example, to print the `enteredName`.
And here you will see that this will print the value
that has been entered because it's set here by calling `save`
because that executes this `onSaved` function
where we do set the value and therefore
in the next line thereafter, `enteredName` will be available.
Hence if I save this and reload,
you will see that if I go back here and I enter test
and I click add item,
if I then take a look at my debug console,
I see my value here
and that's how you can get saved access
to the entered values.
Now of course we can repeat this for the `enteredQuantity`,
but here I wanna get a number
and I'll have an initial value of one.
And we cannot use this quantity to set the initial value.
We'll just have to make sure that we call `toString`
to convert this to a string.

But even more importantly, we can then also use it to store the entered value once save is called. For that we add `onSaved` here as well to this second `TextFormField`, the quantity `TextFormField`. And we get our value here, but this value will be a string. It always is.

The text fields always want and yield strings. Since `enteredQuantity` wants a number, we therefore must convert this value to a number by using `int`, and then here we can use `parse` and `parse` the value.

The difference between `parse` and `TryParse` is that `parse` will throw an error if it fails to convert the string to a number, whereas `TryParse` yields null. But since we already checked whether `TryParse` yields null here in our validator, we can safely use `parse` here because as mentioned before, `onSaved`, this function here will only execute if validation succeeded.

Otherwise we would not call `save` here in our `saveItem` method. Therefore it should be safe to use `parse` here. And we also know that value won't be null.

Hence we can add this exclamation mark because we also check for this in our validator. And with that we're also setting the `enteredQuantity`. Now what about the `DropDownButtonFormField`? Indeed there we do have a `onSaved` method, but here we'll have

to manage the selected value manually anyways,

also to make sure that an initial value is shown because `DropDownButtonFormField` doesn't support an initial value parameter.

Instead it just has a value parameter,

which must be updated every time

the selection changes, though,

so that what we see on the UI

is in line with what's stored internally.

Therefore here as we did it earlier in the course,

in another section, we have to add a new variable,

for example, `selectedCategory`.

And we can set this to an initial value like categories

using this categories list here from `categories.dart`,

and then there the category with an identifier

of `categories.vegetable`.

And for that we have to make sure that we

select it like this and automatically add the import

to the `category.dart` file here from the models folder.

And then here we should also add an exclamation mark because

otherwise Dart thinks that this here could be null,

but we know that it will never be null

because we know that a category

with this identifier does exist in this categories list.

This then also helps Dart infer

that selected category will be of type category.

And now we can update selected category

whenever the selected dropdown value changes.

This can be done with help of `onChanged`,

which is triggered whenever a new selection is made.

And here we get the selected value

as an input automatically by Flutter.

And therefore here we can set selected value equal to value.

And here we should also add an exclamation mark
to make it clear that this will never be null
and to make sure that this is never null,
we should also set value to selectedCategory
so that initially it's this category here
and therefore we do have an initial non-null value,
and thereafter it won't be possible
to select nothing anyways.

You have to select something in that list.

So value should never be null, therefore.

That's why we can then update selectedCategory.

But here we also need to call setState

and make sure that the build method is executed again.

This is required because selectedCategory is used

to set the currently visible value,

and of course that should stay in sync

with what we selected in the dropdown

and should update whenever we change our selection

so that the currently selected value

is reflected on the screen.

That's why setState is needed here.

But with that, we're also storing the selectedCategory,

and that's also why we don't need to add onSave here,

even though we could

because we are already storing the selected value

and we are updating it whenever the selection changes.

So there's no need

to implement the onSave function

on this DropdownButtonFormField.

And therefore here, ultimately we got

all the values we need.

We can now print enteredName, enteredQuantity,

and also print the selectedCategory.

And if we do all that and save that

and force a reload, you will see that if you go here

and you enter something and maybe select dairy

and then click add item, nothing happens here yet.

Of course, this will also change later,

but if we take a look at the debug console,

there we do indeed see the values we just selected.

12. Passing Data Between Screens

So now that we can get safe access

to the entered and selected values,

we of course wanna make sure

that those values are in the end

passed back to the grocery list widget

because I actually wanna manage

my list of grocery items

that should be displayed on the screen here.

At the moment, I am of course

using those dummy grocery items

from the dummy_items.dart file,

but that's now about to change

because here in grocery_list.dart,

instead of using these dummy items,

we now wanna manage a variable

or actually a final property

that could be called grocery items.

And this should be of type List,

to be precise, a list full of grocery items

using our GroceryItem class as a type.

And for that, of course,

this import here must be added.

And initially this will be an empty list,
but of course it's not going to stay
an empty list.

Instead, it should change
whenever we added a new item.

But of course, this new item data
is entered on a different screen.

So how do we get the data from this screen
to this first screen
from which we navigate to the new item screen?

Well, by using a Flutter approach
and feature about which you learned
in the navigation section.

You can pass data back from screen B to screen A.

And you can do this in screen B
so the screen to which you navigate it
from screen A
by using the navigator class
that's built into Flutter of context,
and calling pop.

This pops this screen off the stack of screen,
and navigates the user
back to the previous screen
just as the back button on the device
or in the app bar does.

But we can now pass some data to pop here.
The data that has been entered and validated,
and the data that I do wanna pass along here
is a grocery item.

So I'll create a new grocery item here
using my grocery item model
from the models folder.

And here we then must set an ID name and so on.

Now for the ID, I'll generate a new ID

by using the built-in daytime clause,

and then the now constructor function.

And then I'll convert this to a string.

Technically, this is not a perfect unique ID,

but it's good enough for this dummy app,

and we'll switch to a better ID later

once we add a backend to this application.

So for now, it's good enough,

and we can continue with the name.

Now for that name here,

we of course wanna assign the entered name,

which we extract

with help of save as you learned.

For the quantity,

it's of course the entered quantity.

And for the category,

it's the selected category like this.

And this will create a new grocery item,

and pass it back to the screen

from which we're coming,

which is this grocery list screen.

And there we can get access to the data

that's being passed back

by using async,

and then await and get back

the new item like this.

That's what you learned

in the navigation course section.

Push yields a future that holds the data

that may be returned by the screen,

which you pushed onto your stack of screens.

And it's a future because of course

it will take some time

until the user is done

on this new item screen here.

But eventually we'll get back that data.

And that data here should be

of type grocery item.

Push is a generic function

where we can add this annotation

to tell dart which kind of data

will eventually be yielded.

This makes sure that dart understands

that new item here is a grocery item or null,

and it may be null

because of course we can also come back

from this screen here

by using the device back button

or the back button in the app bar.

And in those cases,

no new grocery item will have been created.

That only happens

if we clicked this add item button,

which triggers the safe item method.

and we then passed validation and call safe.

Only in that case will a new grocery item

be created and passed back.

That's why new item could be null here,

and we should check whether it's null.

If new item is null, I want to return,

and do nothing else in this add item method here

because there is nothing left to do

if we got no new item.

If we do have a new item though,

then it should be added

to this list here

because now of course we have a new item

that should be displayed on the screen.

And this can be achieved

by using the grocery items list and calling add.

And this then allows us to add

the new item to the list.

Now this should be wrapped in set state here

because of course we want to execute

the build method again

because now I want to use grocery items

here in my UI.

Instead of using grocery items

from the dummyitems.dart file here,

I want to use my `_groceryitemsvariable`

or property which we just created,

and to which we add a new item eventually.

And I want to use that for deriving the length,

but then of course also

for accessing the individual item by index.

So in all these places here,

it's `_groceryitems`

and we can therefore,

and also get rid of this dummy items import here,

because we're not using this anymore.

We could also delete the entire file

if we wanted to like this.

So now we're using our locally managed list,

and a new item should be added

if we submitted the form successfully.

Therefore, now if I reload this,

we can give it a try.

We see nothing on the screen initially,

and we can go to this screen

and for example, add milk,

one milk like this.

And indeed we can see it here.

And of course I can also add another item

here like this.

Of course, I got a typo here,

but it does all work as it should.

And with that, we're using this form widget,

and all the form related functionalities

to add new items to our list.

Now for some fine tuning,

I just wanna make sure

that if we got no items yet,

we show some fallback text

centered in the middle of the screen.

For example, a text where we say,

"You got no items yet."

Or anything like this here.

And I wanna make sure that of course,

once we do have items,

we can also get rid of them

so that we can swipe them away,

and delete them.

And definitely take this as a challenge,

and an exercise for you,

and try adding those features on your own,

before we then add them together

in the next lecture.

13. Final Challenge Solution

So, let's now solve this challenge together.

And for this, I'll start with the fallback content

if we have an empty list.

For this year, in `grocery_list.dart`, in this build method,

I'll add a new variable,

a new local variable that only exists in here,

which we'll name `content`,

which initially should contain my fallback content,

which in my case here will be the center widget

with a child of some text.

So, a Text widget where we could, for example, say,

No items added yet, or anything like this.

And this can and should be a const widget.

So, that's the fallback content.

But now I wanna overwrite and change content

if my `groceryItems` list here is not empty.

So, if this is not an empty list,

I wanna override content to hold that list view.

So, in that case, content is set equal to,

and then it's this list view here in the end.

So, we can cut this

and use this as a value here

and add a semicolon at the end.

And now, I got the list view in content,

if we have list items.

And then of course,

we can and should use content here

as a value for body.

And that's how we can show our conditional fallback text.

With that, if I restart, I see my fallback text here,
but if I then
do add a new item here,
that goes away,
and I see the list instead.

Now let's make sure that these items are also dismissible
and go away if we swipe 'em away.

And in an earlier course section,
you saw an example for how this could be implemented.

We can wrap our ListTile here,
which is responsible for outputting
a single row in that list with another widget.

And that other widget should be the dismissible widget,
another widget built into Flutter.

This Dismissible widget then wants a key, as you learned,
to uniquely identify every list item.

And such a key can be created
by using Flutter's ValueKey clause.

And to ValueKey,
you must pass some unique identification criteria,
some unique value for that list item.

And a perfect unique value here, of course, is the ID,
so I'll pass that as a value to ValueKey.

And with that added, we can then also add onDismissed
to register a function
which receives the swipe direction as an input,
and in which we can then get rid of that item.

So, for that here, we could then reach out to our item here
or add a new method,
removeItem, where I expect to get
my grocery item as an input,
and where we then can call setState

to re-execute the build method and update the UI.

And then here, we can use `groceryItems`

and `removeItem`, like this.

And now we just have to execute `removeItem` inside of here

and pass the item to `removeItem`.

And of course, the item is identified

in the `groceryItems` list by its index.

And that's all it should take

to make these items dismissible by swiping them away.

If we save that and reload,

we can add new items here like Milk,

or again, Toast, this time without a typo.

And I can now swipe Milk away,

and it's gone.

I can add Bread instead.

Maybe two.

Like this.

And I can swipe Toast away.

So, this all works.

I see the fallback if nothing's left,

and that's the finished shopping list.

14. Module Summary

Now with that,

we built this shopping list app,

and we practiced many of the concepts and features

about what you learned earlier in the course already,

like passing data back with `pop`

or using widgets like the `ListView.builder`,

rendering content conditionally,

using the `Dismissible` widget, and so on.

But most importantly, in this section,

you learned about the Form widget,
how you can connect a formKey to it,
which is a GlobalKey,
and how you can then use this formKey
to trigger validation,
the save method to store the entered values in variables,
and how you can, therefore, use this Form widget
combined with other special FormField widgets
to conveniently manage possibly more complex Forms
and Form input combinations.

Adding validation and showing error messages
is a breeze, as you learned.

And getting hold of the entered values
also is really straightforward.

And therefore, you might wanna consider using Form,
especially if you are building
more complex user interfaces.

11_shopping-list_connecting-bacend

1. Module Introduction

Now that we learned a lot about building Flutter apps and working with user input, we're now going to learn how we can connect our Flutter apps to backends. And for that, of course, you're also going to learn what exactly a backend is and why you might want to add such backends to certain Flutter apps or under certain circumstances. And you will learn how your Flutter app can communicate with such a backend, if you need one, how you can send HTTP requests from your Flutter apps to your backends.

2. What's a Backend? And Why Would You Want One?

So for this section, let's start with the most important question. What exactly is such a backend and why would we need one? Well, at this point in the course, we, of course, built many Flutter apps, but there's one thing all these Flutter apps had in common. They, of course, run on the user's device, their mobile phone. Therefore, any data that's generated by that app, like for example, that list of grocery items, any such data is stored locally on that device. And therefore, if that device would be lost or replaced or erased, all the data stored on it would be lost and erased as well.

Now, for such a shopping list,
that might not really be a problem,
but of course for other kinds of apps,
that might be a huge problem.

Another potential problem is that some apps actually need
to share data with other users.

I mean, the Uber app wouldn't be
that helpful if only you could use it.

Of course, your app must communicate with potential drivers,
and those drivers in the end communicate back to you.

And the same is true for many other apps.

The Amazon app, of course,
would not work if it could only work with data
that's stored locally on your device.

Instead, the Amazon app needs to be able to reach out
to Amazon's servers to load the data from there.

And that's the entire idea
behind adding such a backend server.

The idea is that your app does not need to run
and work alone,

but instead in conjunction with some services and data
provided and stored on such backend servers.

So these servers are really just computers machines running
somewhere in the internet.

And of course, the exact configuration
and which features a certain backend offers
depends on the backend.

Maybe you built it yourselves,
maybe you wrote your own backend logic,
maybe you're using some third party solution.

So that depends.

But in the end, they will very likely

also store data in some database

besides doing other work, which is also possible.

And since they do run in some data centers

somewhere in the internet, of course,

app users from the entire world can interact

with the same data since it's stored in one central place,

instead of on the mobile phones of your users.

Of course, your flutter app must be able to communicate

with those backend servers to use their services and data.

And that communication typically happens via HTTP requests.

And that is exactly what we're going to take a closer look

at in this course section.

We're not going to build a custom backend,

because this course is not about backend development

or using other programming languages.

Instead, we'll use a very simple dummy third party service

as a backend, but you will learn

how you can then send requests to that service

to store data there or retrieve data from there.

And that, of course, will then also allow you to use

that knowledge that you do gain in this section to adapt it

for other use cases where you might be using other backends.

3. What Is HTTP & How Does It Work?

So for certain kinds of apps,

a backend server might be needed.

And as mentioned in the previous lecture,

the communication between your Flutter app

and such a backend server happens via HTTP.

Now, in case you don't really know how HTTP works

or what HTTP requests are, here's a quick overview.

If you do know what HTTP is,

you can of course, skip this lecture.

So HTTP is that thing that in the end allows us to connect frontend and backend.

It's in the end, a standardized protocol that's used for communicating back and forth between a frontend and a backend.

And here's how that works.

It always starts with the frontend sending a so-called HTTP request to the backend if it wants something from the backend.

For example, in that shopping list app, when a new item is added, you might want to store that on a database that's managed by the backend, and therefore you would then send such a HTTP request from the frontend to the backend, telling the backend that the data should be stored.

The backend then does its thing, it does the thing you want it to do, and then ultimately, it will send back a response.

Though when I say it does the thing you want it to do, that of course, implies that the backend is prepared for the kind of work you want it to do.

It's the code on the backend, the code with which the backend is written that in the end defines what the backend is able to do and which kind of HTTP requests it accepts.

In this course section, we're going to use a dummy third party backend, a service that offers various backend services.

Alternatively, you could also write your own backend by using a programming language like Node.js, for example.

That's not what we'll do in this course,
but that is what you could do.
But ultimately, if you send the request
that's supported by the backend to the backend
and the backend then processed that request
and did whatever it had to do,
you get back such a response.
And both request and response are in the end,
standardized information packages.
This HTTP thing is a standardized protocol
that in the end defines how requests
and responses may look like.
For example, requests typically contain a URL,
the address of the backend
to which the request should be sent,
an HTTP method that can be picked up by the backend
to perform different actions
for different requests with different methods.
You could also add headers to your outgoing requests,
which in the end is just a bunch of metadata
that could be picked up and used by the backend.
And you can also attach data to your outgoing requests,
like for example, a new grocery item
that should be stored in the database.
That could be data attached
to a request sent from the frontend to the backend.
Now when the backend then is done handling the request,
it will send back a response, which then in turn can be used
by the frontend to update the UI
and that response may include headers, so metadata again,
and also a body.
So data that is generated by the backend

and that's then sent back from the backend
to the frontend through the response.
So that's the first overview of how HTTP works.
If you wanna learn even more, you can search
for MDN HTTP to find the MDN Web Docs,
which is a great resource
for learning more about web development
and how the internet works.
And ultimately, this front
and backend thing is an internet thing,
even though we're building a mobile app,
but we're using the internet for communication
between the mobile app and the backend.
So here you can learn more about HTTP if you want to.
And with that, you now got all the base knowledge you need
to get started with this section where you will learn how
to connect your Flutter app to a backend.

4. Setting Up a Dummy Backend (Firebase)

Now in this course section, as mentioned,
we're going to use a dummy third-party service as a backend,
and we're going to use Firebase,
which is a service offered by Google.
But you could, of course, use any backend.
This is just one example for this course section.
And the advantage of Firebase is that it's a combination
of services and database features that you can easily use
and with which you can get started for free
as long as you have a Google account.
And we can use this Firebase service
to practice sending requests from Flutter to such a backend.
That's why we're going to use it,

and we can get started with it
by simply searching for Firebase and visiting its website.
And then there you must log into your Google account
so that you then can go to their console.
And once logged in, you can create a new project.
Now you can give this project any name of your choice
and then click continue.
You don't need to enable Google Analytics.
And with that, this new project gets created.
And I wanna mention right away
that Firebase actually offers many services
which we're not going to use here
which could be useful for your projects,
but we're not going to use them here.
But instead we're just going to use it as a dummy backend
because Firebase thankfully makes it quite easy
and straightforward to send requests to it
and to store data in a database
by sending certain kinds of requests to Firebase.
That's why we're using it here for this demo.
So with the new project created,
you can then go to build here on the left
and there to real-time database,
and that's the service we're going to use here.
So it is a service that allows us to send HTTP requests
that are formatted in a certain way to certain URLs
to store data in this Firebase managed database.
Now, the URLs to which you can send requests
to store data can be found if you search
for Firebase real-time database REST.
And this will lead you to the official documentation
of the so-called REST API,

which is a documentation of the URLs
you can send requests to
and how these requests should be configured to, for example,
get data, store data, and so on.
Now we'll see all of that in action.
I'm just showing this here so that you know
where all this knowledge is coming from
and how you could dive deeper.
Here we just need to create such a real-time database
and the region here does not matter.
And now under security rules,
it's important that you start in test mode.
This will ensure that we are able to send data
and get data from this API without any issues.
If you were using this for a real app,
you would wanna start in locked mode
and you would also wanna learn more
about authenticating requests,
but here we'll start in test mode.
So with that, we got the backend set up
and we can now come back to our Flutter app
to send requests to that backend from there.
And here, I'm again going to use this shopping list app
from the previous course section.
But of course, sending requests with HTTP
to a backend could be done from inside any Flutter app.

5. Adding the http Package

So let's get started with sending HTTP requests
to this dummy backend,
which we configured in the previous lecture.
And then we'll get started in the `new_item.dart` file.

There we have this `saveltem` method
where we validate the form and get all the entered values.
And at the moment we're then moving back
to the previous screen
and we're passing the newly added data along.
And this is absolutely something we can do,
but at the moment, of course, the data would therefore
only be stored locally on the device.
And actually not even that.
At the moment it would only be stored in memory,
to be honest.
If you would wanna store it on the device,
you would have to take some extra steps
about which you will also learn a little bit later
in an extra section, but at the moment,
we would only store the data locally in memory.
And this might be enough for some apps,
but here we're going to practice
working with HTTP and backends
and we're going to send that data
to a backend instead to this firebase backend here,
to be precise, to make sure
that it's getting stored there
in this Firebase realtime database in this case here.
So how can we do this?
Well, first of all, we need to be able
to send HTTP requests,
and this can be done with an extra package,
which you can add to your Flutter project,
and that would be the HTTP package.
If you search for Flutter HTTP,
you should find its site here on pub.dev.

And there you can of course learn more about this package
and how it works
and also how it should be installed
by running this command here essentially.
And you can also read the README thereafter
to learn how it's used,
but, of course, we are going to use it together.
So here I'm running `flutter pub add http`
to install this package.
And with it installed, we can get started with it.
For this here in `new_item.dart`,
we can import this `http` package
and then from there, the `http.dart` file.
And I now recommend adding something
to this import statement,
something which we haven't added before,
but which is useful for this package here.
I recommend adding as `http`.
Now the `AS` keyword here is an important keyword
and in the end it tells Dart that all the content,
that's provided by this package, should be bundled into,
you could say an object, of this name here.
So in this case `HTTP`.
But this could be any name of your choice.
But `HTTP` makes a lot of sense
because it's a lot of `HTTP` functionality
that's exported by this package.
I'm doing this
because this makes interacting with this package
and the functionalities provided by that package
simply a bit easier.
And with that added, we can now use it here in `saveItem`

to send a request after validating and after calling save.
So once we do have our data available,
and we will send this request
with help of this HTTP object
like thing which we created here with help of AS.

6. Sending a POST Request to the Backend

So how can we now send such a request
to the back and to save data there?
Well, as mentioned, we'll use this http object, so to say.
And with the dot notation,
we can now get access
to a broad variety of features
provided by the http package.
Most importantly, a couple of methods,
which you see down there.
Now, these methods allow you
to send different kinds of HTTP requests.
Because when you send an HTTP request,
you can configure it,
and most importantly,
you can choose which HTTP method should be used.
And you also find this
in the official Firebase documentation.
Now, which method does what
will always depend on your backend.
Some backends accept post requests for everything
because in the end, it's the backend logic,
so the code written on the backend
that decides what happens
if a certain kind of request arrives.
Here, in Firebase case, it's quite logical.

And for example, sending a GET request gets you data,
sending a DELETE request deletes data,
sending a POST request pushes new data to the server,
so stores new data there.
PUT would overwrite existing data
and PATCH would update parts of existing data.
So therefore here, of course,
if you wanna store new data,
we wanna use this post method here,
which is provided by the http package
because this sends such a post http request,
and that's exactly what Firebase wants to store data there.
Now, to make this work,
this post request first must be configured though,
and for that, post accepts multiple arguments.
It always wants a url, which is of type Uri,
which is a special kind of object,
and I'll show you how to create
that object in just a second.
That's the url to which the request should be sent.
In Firebase case,
for post, the URL looks something like this,
https, then your PROJECT_ID,
then this part here,
and then this part, which is also dynamic,
end it with .json.
And I'll show you how to construct the url
that belongs to your project over the next minutes.
So that's the url.
But in addition to the url,
post, for example, also accepts a map of headers,
which is in the end just some metadata

that can be added to the outgoing request,
and a body object,
which defines the data
that should be attached to the outgoing request.
And that will be important here
because, of course, a post request
almost always needs data attached to it
because it is about sending data to a backend.
And of course, that data therefore
then also must be part of the request.
So here, we'll start with the url,
and for that, we can add a new url
local final variable here,
which can be created by using the Uri class,
which is built into Dart.
And Uri offers many special constructors,
for example, an https constructor to create a url
that points at a https backend,
so at a server that uses the https protocol.
And that's what we need here
because Firebase wants a request
that's sent to https and then something.
Now, that something part is the next part.
Here, that should be firebaseio.com
preceded by your PROJECT_ID.
Now, to get the full URL up to this path,
you can simply go to your Firebase project
and there to the real-time database,
and it's in the end now simply this thing here,
which is your URL,
so we can copy this
and paste it in here

and we just have to remove the `https://` part
because that will be added automatically
thanks to us using this `https` constructor.

Now, with that, we would be creating such a URL,
but this is actually not the complete URL.
Instead we should add a second argument here,
which now defines the so-called path,
the part after the domain, so to say.

So what you would typically add with a slash
after domain domain.

And here we got a special feature of Firebase,
we can basically pick this path ourselves.

For example, we could set it here
to `shopping-list`
like this.

And the only thing we have to add here in Firebase case
is `.json` at the end.

This is not Flutter specific,
instead this is something required by Firebase.

Now, this part here, `shopping-list`,
is totally up to you,
and it will basically create a sub folder,
a node in this database you could say of that name,
and you will see it here once we started adding some data.

So that's now the path I want to add here.

With that edit, we got the URL
to which the request should be sent,
and now we just have to finish configuring this request.

For example, by also setting the headers named `argument`
to a map where the keys are header identifiers
and the values are settings for those headers.

And here it's just one header, which I wanna add,

Content-Type written like this
with a value of application/json.
This will help Firebase understand
how the data we're sending to it will be formatted.
Last but not least, it's the data itself that's missing.
For this, we should add another named parameter to post,
and this is now the body parameter
which defines the data that should be attached
to the outgoing request.
And here that should in the end be our GroceryItem here,
though I'll format it a bit differently
because body here actually wants data
that's formatted as JSON,
which is a special text-based data format.
Creating data in that format
is quite straightforward in Flutter and Dart
because there's a built-in JSON utility object
that is provided by Dart under the convert sub package,
which you have to import.
But with that added, it is available here,
and you can now use JSON to call the encode method,
and this converts data to this JSON-formatted text.
So it simply encodes the data
to use this special format.
For that, encode needs an object
that can easily be converted,
and here a map is a great choice
because JSON encode knows how to convert maps to JSON.
Therefore, here I'll create a new map object on the fly
by using these curly braces.
And here, I want to encode my key product data.
And in my case that's all that data down there:

name, quantity and also the category.

Though, for the category,

I'll also make a change soon.

But here I'll start with the name,

the quantity and the category.

And in a map, the keys must be wrapped between quotes,

so that's what I did here.

And for the category,

I'll actually just store the title

because otherwise it would be a complete object

where encoding it might fail,

and I'll therefore just store the title

because I can then later use that title

for mapping it to the actual category object

when we then later load the data from Firebase again.

So for sending the data, I'll just store the title

because I'll be able to later use that title

as a category identifier,

and you will see how it's getting used

once we start loading the data.

And that's therefore now the data

that should be sent to Firebase.

We don't need to send the ID

because Firebase will generate a unique ID for us,

which is pretty convenient.

Hence, this is all the data that needs to be sent.

And at the moment, I will also not navigate back

and I will not pass any data to pop.

Instead, I just wanna send this data.

And if we now save everything with these settings

and we restart the app,

we should be able to enter some data here.

And if you now click Add Item,
here in the app, nothing happens,
but if you go to Firebase,
you should now see a new node
in your real-time database here,
in my case called shopping-list
because that's the key I used.
And if you expand this node,
you see a new unique ID that was generated.
And if you expand this item,
you now see the data that was submitted,
and this proves that sending the data succeeds.
Now, if it fails for you,
definitely take a look at the debug console
and the error message you might be seeing there.
Make sure that you are using
your Firebase real-time database ID
and not mine
so that you use your URL, not mine.
And make sure that under Rules,
you don't have false here,
but something like this.
Or simply replace this with true,
this will also allow all incoming requests,
and that's what we want here for this demo backend.
And with that, we are able to store data here on Firebase.
Now, let's also see how we can get the data back.

7. Working with the Request & Waiting for the Response

So how can we now get our data back
from Firebase to our app?
Well, for that, of course, we wanna go back

to the grocery list screen to this widget.

And there we wanna send a get request to Firebase

because I mentioned before

that in the end, it's such a get request

that gets you data from there.

But before we do that, we have to do one other thing.

We have to make sure that we wait

for this post-request to complete.

Otherwise, if we do go back too early, we might not be able

to fetch the data yet

because it might not have finished sending to Firebase yet.

It might not be stored there yet.

Because with HTTP requests, you have to understand

and keep in mind that they take some time,

not a large amount of time typically, but a few milliseconds

or seconds because after all, a request

must be created, must be sent to the backend.

And the backend then must handle that request, do its work,

for example, store the data in a database

and then the backend needs to create a response, a success

or failure response and send back that response to your app.

And in the end, it's this response we wanna wait for

because then we know that everything worked.

Or if it failed, we know why it failed

because we only know whether everything worked

and that the data was stored once we have that response

and it's only thereafter that we should try loading

or manipulating that data on that server.

So that's therefore the next question, how can we wait

for that response?

Well, the post-method of this HTTP package

actually returns some data.

It returns a Future that eventually resolves
to a response object
and you learn about futures earlier in the course already.
Futures are these objects
which will eventually give you access to other data.
And it makes sense that post yields such a future
because we don't have a response instantly
after executing that code,
but instead only once the server is done with its work.
That's why post gives us a future
that will eventually give us access to the response.
Now, as you learned earlier, there are two main ways
of getting access to that response data here in this case.
You can add a then method here
on the result of calling post,
so chained on the post-method in the end,
and then you can pass a function to then
and in here, you would get your response
and then here, you could work with that response.
Alternatively, since this can be a bit inconvenient
to write, you can convert saveItem to an async function
by adding the async keyword here.
And this then allows you to await this response
by adding the await keyword here in front of the method
that yields the future
because a weight is meant to be used on futures.
And then with a weight added here,
Dart will basically add this then method
for you invisibly behind the scenes you could say,
and it will give you access
to the response data once it's there.
So then we get our response here

and we'll wait for this request and response cycle to finish
before this data is available and
before the next line of code will be executed by dart.

And typically, this `async await` syntax
is a bit more convenient, which is why I'm using this here
instead of chaining this then method,
which you could alternatively also do though.

So it's here where we do now have access to this response.

And this response object now has a couple of properties
that can be helpful.

For example, a `statusCode` property, which is an integer,
which tells us whether the request succeeded or not.

There are some standardized statusCodes you can look for.

For example, 200 or 201 tells you that everything worked,
400 ish or 500 ish errors like 404 tell you
that something went wrong.

And you might have seen a 404 error before
if you visited a website that doesn't exist.

So this `statusCode` can be used
to check whether everything worked.

And I'll get back to this later.

For the moment, we'll just assume that it worked
and it then still this response which has a body
with some response data that might have been attached
to the response sent back by the server.

And it will always depend on the backend
and your server, which exact data is stored in that body
and how it's formatted.

In case of Firebase,
we can take a look at the official documentation
to see what's sent back
and we learn that we get back this JSON data here

in a response.

This is how JSON data looks like, by the way.

It looks a lot like a map in Flutter.

And indeed, we can easily convert this back
to a map if we wanted to.

So it's this kind of data we get back from the response.

And in the end, as you can see,

it's simply a key named name which then contains
the unique ID that was generated automatically.

And this can be useful

and this is something we'll use soon for the moment,
I don't care about it though.

I only care about the fact that we do wait
for the response to be there.

And that's what we are doing here thanks to await.

So for the moment here, I'll just print response body
so that we can see it in the console
and I'll print my statusCode.

And with that, we can then pop this screen and go back.

Now here, I'm actually getting a warning

that I should not try

to use the build context across async gaps,

which simply means after awaiting,

because Flutter doesn't know

with certainty whether the context is still the same
as before we awaited.

If we, for example, navigated away from this widget here
in between.

So whilst the HTTP request was on its way,

a context would be referring to a widget

that might not be visible anymore.

That's why it's not recommended

to use context here like this.

Now, to make sure that we're not referring to an outdated context here.

So for example of a widget that's not part of the screen anymore, we can add a if check here and check if this context here is not mounted anymore by using the special mounted property provided by that context object and checking if that's not the case.

By adding a leading exclamation mark, which defines a condition where we want this to not be true because mounted does yield a Boolean, true if the widget is mounted, if it's still part of the screen, false if it's not mounted.

And with that, we're checking for it being false.

And if that is the case, we can simply return here so that this code does not get executed if the context is not mounted anymore.

So in the end, that means that if this widget to which this context belongs is not part of the screen anymore, we return and we don't execute this code.

Now here, I'm actually still getting that same warning, but this is actually a fine way of using context and it is an officially recommended workaround.

So with that, we will now pop this screen if we still can, if it still is visible once this request succeeds.

So this will then take us back to the grocery list screen after we successfully sent our data to Firebase and we get back a response.

So if you now save this and reload the app and I add bananas, 12 bananas, for example,

you will see that if a click add item,
it takes a short while, maybe just a couple of milliseconds,
but it only navigates us back to this screen
after that time span.

Now here, we don't see the items yet
because we're not loading them yet
and we're not passing them back manually anymore.

We'll change this soon,
but we can of course see the bananas here in our backend.

So that works.

And now we're waiting for that response
before we go back to the grocery list screen.

And therefore, now as a next step,
we can finally load the data there
because we now know that we only reach this screen
after the post-request has been sent to Firebase
and the data has been stored there.

8. Fetching & Transforming Data

So now in the grocery list dart file,
we can finally start fetching our data from Firebase.

And here I wanna fetch my data in add item,
after we came back from the new item screen.

Hence, I will keep a weight here,
to wait for this future to resolve,
so to wait for the user to come back,
but we won't get the new item anymore,
because in new item dot dart,
I am not passing any data to pop.

Hence, we shouldn't wait for this here.

We don't need this check anymore and we don't need this code
because I am not getting any new items here,

instead I just wait for the user to come back
to then fetch my data from the backend again.
Now to do this, we need to send such a GET request here
to Firebase.
And this is again done by using this http package,
which are there for import into grocery list dot dart
as I imported it into new item dot dart.
And then here in add item,
after we came back from the navigation,
we can and should create a new URL
in the same way as before.
So here in new item dot dart,
I wanna grab this,
URL construction logic here,
like this,
and with that we can send such a GET request
by using this http package.
And then there instead of the post, the GET method,
this then wants such a URL, and that's basically it.
We could also add such a headers map here for extra metadata
that should be added to the request,
but here, this will actually not be needed.
But we for example, can't add a body
because a GET request is always about getting data,
not sending data, hence you don't add a body to it.
So therefore that's all we have to do.
And now of course we should await this
to get hold of the response
because it will be that response that contains the data.
At the moment we just sent the request,
but of course that's worthless
if we don't use the data that's returned.

Now to see which data is returned,
I'll simply print response here.
If I now save that and reload, this app reloads,
and if we now take a look at the debug console here,
we see nothing.
So looks like no http request was sent,
and indeed, that's the case
because I'm only sending one here after adding a new item.
But actually, I also wanna send such a request
if the app loads up and I reach this screen
for the first time.
Therefore, I'll actually add a new method here,
load,
items,
could be the name,
of course, it's up to you, and the return value is void,
not final.
And it's now in there, where I wanna send my request
and print the response.
And this should therefore, also be a async
method, by adding the async keyword here.
It's now load items, which I wanna execute here
in add item like this,
and we can await it, but I'm not returning any data here
and I don't wanna do anything thereafter,
so we don't have to.
And it's now also load items which I wanna execute
if this widget loads up for the first time,
so if the app just got started.
And how can we execute code if this was
rendered for the first time,
if this widget was just created?

Well, since we're in a state clause here,
we want to add the init state method,
because you learned before, this is a method
that allows you to do some initialization work.

And the initialization work I wanna do here
is that I sent my request.

So here we can call load items, like this,
and this should therefore, make sure that this method,
this method here, is triggered,
and this request is sent if this state object is created
for the first time.

Which is the case if the app just started or restarted.
Hence with this added here, if we restart the app again,
nothing happens here of course,
because we're not updating the UI in any way,
but if I open my debug console,
I do now see this part here.

We got back an object which was of type response.
And if we dive into this object,
by printing response dot body instead,
we should see even more.

If I now reload, you see I'm getting back this data here,
and that in the end is JSON data where I have keys
that are the unique IDs that were generated by Firebase.

And then as values, I have nested maps,
so nested objects, you could say,
with the data that's stored under these IDs on firebase.

So that's the data we're getting back from Firebase
and that's of course the data we can now use
to update the UI.

For that here in load items, we wanna convert this JSON data
back to regular dart objects, to dart maps,

with which we can work,
by using this JSON utility object again,
which is imported from dart:convert, as you learned.
And this now has a decode method.
Before we used N code to convert a map to JSON,
now we use decode to convert this JSON text format
back to a map.
And here I wanna decode response dot body.
This then, yields as a dynamic value
because of course dart doesn't know
how this response body's formatted,
but we know that in the end it be our list data.
And of course this variable name is up to you.
Now this list data will have this format here,
just converted to a map, but the same keys
and the same nested maps.
Of course, I now want to convert this back
to a list of grocery items.
And to do this we could add a for loop here,
where we loop through our,
list data entries,
and we know that list data will be a map.
So we can also tell dart that this will be a map,
and that it will actually be a map where we have string keys
and then map values,
because we'll have these keys here
and then maps as values for these keys.
And these inner maps will also have string keys,
but they will have dynamic values.
We have two strings and one integer.
So here we actually have dynamic values
in those nested maps.

But with that dart knows that list data will be a map
and we loop through all the map entries now.
So all the outer key value pairs with those unique IDs
as keys and the nested maps as the values.
And inside of the loop, we can now convert these nested maps
to grocery items, and we can add them to a new
temporarily created loaded items list,
which should be,
a list of grocery items.
This does not change.
I'm creating a temporary list here
because I then wanna replace this list here,
which we use in the overall clause with this list
after I parsed all the loaded items here.
So in this loop where we now go through all the items,
we wanna add new items to this temporary local list here
by calling add.
And we wanna create a new grocery item here
by using our own grocery item clause
and its constructor function here.
And to grocery item, we now have to pass the ID,
which actually should be item dot key.
So these auto-generated IDs, provided by a Firebase,
I wanna use those as IDs because these are perfect IDs.
And then for the name, for example,
I wanna use item dot value to dive
into this nested map here,
and then there it's the name.
So item dot value, and then with square brackets, the name.
This is how we access this key with a name of name
inside of this nested map.
And now we can do the same for the quantity.

This would be item dot value,

quantity like this.

And for the category, we don't want use item value category

because the category here is just the title,

because that's what we sent with the post request before,

just the title.

I now wanna get a full category object again.

And of course here, I actually wanna get a category

that's stored in my categories list,

so this list of predefined categories.

But of course, thankfully there we can simply search

for a category with a certain title.

Now to get this category from this categories list here,

we can search for it before we start adding

a new grocery item to our loaded items here

by creating a new temporary local category final variable.

And then here, I wanna reach out to my categories map

and there to all the entries,

which as you learned is a list of items

where every item has a key property and a value property.

And now since this is a list here,

we can search for the first item

that matches a certain condition,

which can be done with first where.

And this is of course not something you have to do

every time you're loading data with the http request,

instead, that's specific to this app here

where I sent only my category title to Firebase

and I now wanna match it to some local in-memory data again.

First where is almost like where,

which we used earlier in the course to filter a list.

First wear will now also filter a list,

but it will only yield one item, the first matching item where a certain condition is met.

So here we got our category item or cat item, which is such a item with a key and a value property, and now I wanna find the item where my cat item value dot title, because this will now be one of the categories in my list of categories here, or in my map of categories, to be precise.

So where I have one of those categories where the title is equal to item, dot value, category.

That's the category, in the end, I wanna add down there.

Now again to make this clear what we're doing here, where going through all the entries in this categories map here where our keys are enum values and the values are category objects.

And we're taking a look at all these items and for every item I check of the value, so this category object has a title that's equal to the value stored under the category key on that item that's part of our response, because this list data here, that's our response data.

So item refers to the response data, CAT item refers to the categories in this predefined categories map here.

With that I get back my category here, or to be precise, such a map entry where I then can chain dot value at the end here, and this will now finally give me the category that was identified.

With that, we got the category which we can set

as a value here when creating that grocery item.

And this now creates a new grocery item,
adds it to this temporary loaded items list,
and that's that for all the items
that are part of the response.

So that for every response item we create a new grocery item
with the category from that list of predefined categories
and the data that's fetched from the backend,
including this auto-generated unique ID,
which is provided by Firebase.

Well, and now after going through all of that,
I wanna use my loaded items to override
my locally available grocery items,
so this grocery items property
of the grocery list state clause.

Now to override it, I'll convert this from final to var,
or since I have a type annotation here,
get rid of that keyword
because now I want to reassign grocery items
and I wanna reassign it here.

After this for loop,
so after we close this for loop,
I wanna set grocery items equal to loaded items
to overwrite that list.

And this should be done inside of set state here actually,
to make sure that the build method executes again
and the UI is updated.

With that, if we save this,
nothing happens here.

So let's go back and open the debug console,
and there we learn that the problem seems to be that
the type map string dynamic is not a subtype of map

string dynamic, so my type definition here is a bit too specific for dart. We should simply replace it with map dynamic to make sure that dart's happy with it. Now you may also see a slightly different error message, where you see the type null is not a subtype of type, map something and you'll get this error message if you actually have no items on your backend because you haven't followed along with the previous lectures or because you deleted all the items that have been added to Firebase. To get around this error, you should simply add some items to Firebase, and as soon as you do that, the error will go away and we'll handle the case that we have no items on the backend, a little bit later in this section.

With this adjustment made though, now this loads, and now we can see the data here deleting won't work, but we can see it here.

And if I now add a new item, like milk, we go back after adding it, and we see that data here as well.

Also, of course, if I reload, 'cause then it also loads again.

And that's how we can load data from the backend, but that's not all I wanna do related to that.

9. Avoiding Unnecessary Requests

So now in the previous lecture, we added the logic to fetch the data. And here I actually got a little hint

that I should not name my local variable,
which is only used in this function,
with an underscore at the beginning.

It works, but it's actually not recommended,
so let's get rid of that.

But with that, this is how we're loading the data.

And of course this works.

The data is here as you can see,
but there are actually various aspects
that we can improve here.

For one, it seems a bit redundant to send a request
and load the data if we're coming back
from the new item screen
because there we actually have all the data,
and we even get that ID that's generated by Firebase.

We get that as part of the response body.

If you go back here and add some new item here,
you will see that in the debug console,
you get this data here.

Now the 200 is the response code,
but this here is actually the ID.

So therefore it seems a bit redundant
to fetch the data again,
because we already got the name, quantity and category,
and the only missing piece was the ID
because that's generated on the backend now by Firebase,
but of course we get this here as well.

And therefore it actually would be a bit better
to avoid this extra get request
and instead create the grocery item here in saveItem
as we did it earlier in the course to then add the ID
to it once it's available,

and to then send back that finished grocery item
to the grocery list screen so that there,
after adding a new item, we don't have to call loadItems,
but we instead simply get the updated item
and we can manually add it to our local list
and avoid this extra http request.

And that's indeed what I'll do here.

I will switch back to the code we had before
where I do pass a new grocery item to pop,
and here the ID should be the ID
that's part of our response body.

So this ID here.

Now this response body is in JSON format.

So to access it, we first of all have to decode it.

So here we got our response data,
which we get by using this JSON utility object
and then the decode method.

And here I wanna decode response body.

This will give me a map in the end.

So therefore we can also add this annotation.

This will be a map of string dynamic.

And here my ID will be resData.

And then as you can see in the debug console,
it's stored under a name key.

So the ID is stored under the key name
and hence we can access the key name here.

And this will be our string.

So if that will get the ID,
next, we'll also need to set the name,
and of course, that is still available from up here,
the entered name, entered quantity, and so on,
that's all still available.

So we can simply use these values,
entered name, entered quantity,
and last but not least, of course,
also the category which is set to select that category,
not just the title, but the full category.

Then we pass this data when we pop here,
and the offering grocery list,
we can again expect that data.

So just as we had it before.

Now we can await for this route to be removed
and we'll get back that new item,
which could be null.

So we should still check for that
because maybe we used the device back button
instead of the add item button.

And if it is null, we just return here.

Otherwise, we wanna update our state,
and we wanna add a new item to that state.

So here, I'll reach out to my grocery items property
and add my new item to it just as we did it before.

And with that, we save that extra request.

So now if I save this and reload,
the data still loads when this app starts
for the first time.

But if I then go here and, for example,
buy some chocolate, maybe a bit more like this,
we will still wait for that request,
but then it's added instantly here
without sending another request behind the scenes.

And I'd say that's an optimization
which we definitely should apply here.

10. Managing the Loading State

Now, avoiding this extra GET request

was one thing I wanted to improve.

There is another thing.

If we reload the app, we see no items added yet

for a fraction of a second whilst the request is on its way.

So until the data is there, until we have the response.

And that's of course not a great user experience.

It would be better to show some loading spinner

or anything like that until the data is there.

Instead of these no items added yet text,

which is simply wrong in this situation.

Now to show such a loading spinner,

we can show some conditional content.

For that here in grocery list,

I can get rid of this import,

but that's not too important.

I wanna add a new variable here in grocery list state,

and that would be a variable

that could be called `isLoading`,

which initially could be set to `true`,

because we know that if this widget is rendered

for the first time, we will load our items.

So we can set this to `true` initially,

and I wanna set it to `false` once we have our data.

So therefore down here,

where I set my grocery items to the loaded items,

I now also wanna set `isLoading`.

And I forget the leading underscore here.

So it should be `_isLoading`.

And I wanna set `_isLoading` to `false` here

once I have my loaded items,

because I am not loading anymore at this point.

And now we can use `isLoading`

to show different content on the screen if we are loading.

So here where I'm setting my content to this,

and to this, if I have items,

I can add another if check,

and check if `isLoading` is true,

and we don't need to add `equals equals true` here,

because `isLoading` already contains true or false.

So this is enough.

And if it is true,

I wanna set content to a center widget,

where the child is a circular progress indicator.

And this is a built-in widget

that simply renders a loading spinner.

And that's the content I wanna display on the screen

if `isLoading` is true.

And with this change made,

with this tiny addition made,

you see this loading spinner initially,

of course, only for the fraction of a second

since the response arrives pretty quickly typically,

but this is now a better experience

because a loading spinner

of course makes more sense than this fallback text

which was simply incorrect.

Now the loading spinner we added to the grocery list

is also something we might wanna add to new item,

because there, of course, if I am adding a new item,

like beef,

and I click add item,

nothing happens for a fraction of a second

before we then go back here.

And if the request takes a bit longer,

which also sometimes happens,

we might be stuck on this page

for one or two or three seconds

until we navigate away.

And in all this time,

the user doesn't know what's happening,

and they might even press the add item button again,

which would lead to another HTTP request being sent again.

So, therefore, I definitely want to disable

add item and reset.

I want to disable these buttons

whilst a request is on its way.

And I also wanna show a loading spinner to tell the user

that something's happening behind the scenes.

Therefore, here in the new item file,

we also want to add a new state variable,

isSending, sounds like a sensible name.

And initially that's false here

because when this widget is loaded for the first time,

we're not sending any data yet.

But I wanna set this to true

if we made it into this validate block here

because I know that here we then save the data

and we're sending the data thereafter.

So after calling save, we can execute set state,

and set isSending to true here.

And now we can use this isSending information

to update the UI.

Down there where we have the buttons,

I want to disable the button if we are sending.

And in Flutter, a button can be disabled
by simply passing null to `unPressed` instead of a function.
So here I wanna set the value of `unPressed`
with help of a ternary expression,
which might sound strange
since we're creating a function here,
but keep in mind, in Flutter, functions are just values,
so we can treat them like any other value.
Therefore, here I can check if `isSending` is true.
And if that's the case, using a ternary expression here,
I wanna set `unPressed` to null, instead of this function,
which is my fallback case,
so that this function is used
as a value for `unPressed` if `isSending` is false,
but if it's true, null is used.
And we do the same down here, `isSending`,
and if that is true, we use null,
otherwise, we use `saveltem` here, the function.
So that's the first step.
I also wanna replace the child content of `ElevatedButton`.
Instead of showing this text here,
we could show a text like, "Saving..."
if we are sending,
or this circular progress indicator.
And that's what I'll do.
For this here, we can also check `isSending` down here
with a ternary expression,
and check if `isSending` is true,
in which case, here I wanna render a sized box,
with a height of 16 and a width of 16,
which I simply set to make sure
that this circular progress indicator has the right size.

And then we can set the size here
to circular progress indicator again,
as we did it before.
And the fallback value still is our const text.
And this size box here can also be const.
So now we're setting this here as a value
if `isSending` is true,
and this text otherwise.
And with those changes made, if we now save this and reload,
if I go back here, and I add apples here, five pieces,
if I click add item, you have to watch closely
because it will all happen pretty fast.
But if I now click add item, you'll see
the buttons were disabled
and you could briefly see that loading spinner.
And that's how you can manage the loading state,
or the sending state to be precise on this
add a new item screen, as well.

11. Error Response Handling

Now when sending HTTP requests,
no matter if you are sending data or fetching data,
sometimes things can also go wrong.
And to show you a scenario where things do go wrong,
I'll go back to the grocery list.
And there in load items that can simulate such an error
by using an invalid project ID here.
A, B, C, for example.
If I do that and I save and reload,
you will see that here we're stuck on the loading screen.
Nothing happens.
And this happens

because if you take a look at the debug console
in the end here, I got an error.
Now the exact error message might be different.
It depends on the problem you're facing.
But the main problem here, of course,
is that we're not getting back the data from this backend
because here the URL is wrong.
Now, of course, that's an error
you typically can avoid as a developer.
You can simply test this app and use the correct URL.
I'm just using this scenario here as a demo.
But of course you can definitely face some problems
when sending requests
because, for example, the Firebase servers
could be offline temporarily or anything like that.
Therefore, when sending HTTP requests,
you must be prepared for things to go wrong.
And you can find out if something went wrong
by taking a look at the response status code.
If I print this here
and I reload so that this app restarts,
you can now see in the debug console
that somewhere in here you will find this status code,
in my case here, 404
because it doesn't find a project with that ID here,
as it seems.
Or it doesn't find a shopping list note there at least.
And as I mentioned before,
if status code is greater than 400,
you basically have a problem because all these 400
and 500 codes are error codes.
And here we might wanna check for our status code

to be greater or equal than 400

to then show an error message to the user

instead of keeping them stuck on this loading screen.

At least we don't wanna keep them there without telling them that something's wrong.

Now, you can either check for any error as I'm doing it here

or look for individual error codes like 404,

which you get if a resource has not been found,

to show error specific error messages.

That's up to you.

Here, I'll use the more general approach.

But with that,

we can now introduce a new state variable error,

which might be null

and which should eventually be a string if it's not null.

So that's how we can define it.

Property which may be null,

but which if it's not null, should be a string.

We can set error here if we do have an error status code.

In that case, we could set error to

failed to fetch data.

Please try again later.

Something like this.

The exact error message of course is up to you.

Now with that,

you can then go down to your build method again

and there at the very end,

after all these other if checks,

I want to add another if check

and check if error is not equal to null,

which you can check like this.

And if that's the case, I know that I have an error.

And, in that case, I don't care about what I wanted to display as content.

If it was the fallback or the progress indicator, I don't care about that.

Instead, I wanna render my error message.

And for that, I'll copy this here and set content equal to this.

But now the text here will be the error message stored in error and therefore this can't be constant anymore.

Here we should also add an exclamation mark because we know that error won't be null because that's literally what we're checking here.

So with that, I'm setting my content of this screen to this error message if we have an error.

Of course, now I just must make sure that in load items where I'm setting error, I also call set state because since I'm outputting the error in the UI, I also must make sure that the UI is updated if an error occurs.

And we do that with set state, of course.

With that, if I save this and force a reload, now we get this error message here if things go wrong.

And that's arguably of course, a better user experience than what we had before.

With that, I just want to add a dot here at the end.

I'm happy with that and we definitely improved the app.

Now with that adjustment made, I will of course grab the correct URL again and use that again because I don't wanna break this app.

Without https and this slash here, by the way.
Because I don't wanna break this app,
I just wanted to show you how you could handle errors.
Of course, the default case here is
that we don't have an error.

12. Sending DELETE Requests

So now with all of that,
we can now load our items,
we can add new items.
The missing piece, of course, is that we remove items.
I mean, we can remove items,
but if we then reload, they will be there again
because we only deleted them locally and not on our backend.
That, of course, needs to change.
And for that we should go to this remove item method here
in grocery list.
And there we should send another HTTP request.
Now it should of course be a delete request
sent to a URL that's structured like this.
And there's one special thing about that URL.
It's the same URL as before,
but at the end,
we're pointing at a specific item in our Firebase database.
a specific item by ID.
So we have to include the ID
of the item we wanna delete in the URL, as you will see.
Now for that here, we'll of course start by creating a URL.
So we can copy this logic here from load items,
and add it to remove item.
But now the URL must be adjusted.
Instead of pointing at the overall shopping list,

we wanna point at a specific item in there,
because we don't wanna delete the entire shopping list,
but just one item in the shopping list.
That's why we have to add another slash here
after shopping list
to target a specific item in there.
It's like working with a folder structure.
We wanna dive into the shopping list folder,
and then target one specific folder by name.
So by ID here.
And now the ID here,
of course, depends on the item that was selected.
So here I want to inject a value into this string,
which we can still do,
because this is a regular string.
We can use this injection syntax
about which you learned earlier, and target an item by ID.
And since these IDs are loaded from Firebase,
this should work just fine.
So with that, we're targeting a specific item,
and that will be important
because next we'll use the HTTP package
to send the delete request to this URL.
Now just like get delete,
doesn't need a lot of configuration,
you could add a body technically and you can set headers,
but we don't need to do any of that.
We don't need any extra metadata,
and there also is no extra request body attached
to this request.
Instead we just send a delete request to this URL
and Firebase will do the rest.

I then of course, also still wanna update my local state
to also remove that item from the local list.
I don't wanna fetch the entire list again,
I just wanna remove that one item which we deleted
with this request from the list.
Now we could add `async`, a `wait` here, to do that,
but we actually don't even need to do that.
I'm fine with not waiting for this
before I update my local list,
because the item can be deleted in the background.
I don't need to wait for this request to finish
to update my local list here.
I had to wait when adding a new item
because I needed that ID from the backend.
But here for deleting,
I don't need anything from the response.
And therefore we can delete it instantly
without waiting for the response,
and if we don't add a `wait` here, we also won't wait for it.
Instead, this request will be sent,
but we'll not wait for the response,
and therefore we should be good to go.
With that,
if I save that and reload,
if I then wait for the data to load,
Firestore was a bit slower here,
and I then delete test, it's deleted locally,
but also on the backend
as you can see here.
It's highlighted in red,
which is quite convenient, at least for a couple of seconds
before it's then removed here.

And if I now reload the app,
we also won't see the test item here anymore.
I mean I have it here again,
but if I delete this and maybe also chocolate,
those items will be deleted also from the backend.
And if I then reload, those items will well stay deleted.
Now of course, things could go wrong when deleting.
I could again have an invalid URL
or an error that's beyond my control.
And in that case, we have a problem,
because now I can delete items locally,
but they're not deleted on Firebase
because something went wrong with the request,
and for that reason, we then, for example,
could be surprised if we visit the app again,
and the item is back.
Therefore, we actually might want to check
for whether this request succeeds and undo this deletion,
and maybe show an error message to the user
if we fail to delete the item.
For this, I will convert this
to async so that I can await this,
but I will actually update my list
before sending the request
so that we don't unnecessarily wait here,
but I then take a look at my response here.
And if my response status code is greater
or equal than 400, so I know that something went wrong,
I wanna undo this deletion.
And I can undo it by calling set state again,
and now adding that item back to my list.
Now with add, I would of course add it in a different place,

therefore I might want to get the index first,
which I can get by reaching out to my list,
and they are index of item.

And then here instead of add, we could use insert
to add item at a specific index.

And that's the same approach we used earlier in the course
with a snack bar and the Undo option we had there.

Because insert is a method built into Dart
that can be used on any list, to add an item,
add a specific index in that list.

So that will undo the deletion if we have an error.

And of course, we also might want
to show an error message here
with a scaffold messenger, and a snack bar for example.

But here, I'll keep it a bit simpler and not do that
and just undo the operation.

So now if I delete this, it will eventually come back
as you see because the deletion fails.

But of course it might be better
to also show an error message here.

But since this is nothing new
and not directly related to HTTP,
I'll simply skip this gap here,
but it is of course something you could do.

And with that, I'll of course now fix this URL again
to make sure that deleting does not work again,
and we are using a valid URL.

13. Handling the "No Data" Case

So now with all these new features added,
we're almost done for this section,
but there's one problem with this app.

And to show this problem,
I deleted all items but one, which I added,
and if I now also delete this last item,
you will see that if I reload here,
I am stuck again in this loading spinner state.
I don't see an error message
and I don't see the fallback text either.
And we have this behavior here
because we're actually facing an error here.
An error which you can see in the debug console.
This unhandled exception type Null
is not a subtype of type Map error.
Now we're getting this error here
because in the end, now I have no items left on my backend,
but here in load items,
I'm trying to decode my response.body.
So I'm trying to decode that data,
which I'm getting back from the backend,
and I'm telling dart that the decoded data will be a map
where I have string keys and different kinds of values.
The problem is that if I have no items on Firebase,
response.body won't yield such a map though.
Instead, if I print response.body here
and I restart the app,
you will see that here it's of type null.
This very first line here, it's of type null.
So we got no data back.
And that's why we're getting this error thereafter,
because null, when decoded here, is not giving us a map.
So therefore, here we should add an extra check
and check if response body is equal null,
which means we got no data from the backend.

And in that case, I want to simply return
and not execute any more code.
Though I also want to do at least one thing before I return.
I want to set my state and set isLoading to false.
Because we're not loading anymore,
we're done with that.
We got a response.
It doesn't have any data attached to it,
but we still got a response.
So we're done loading and I therefore
want to get rid of my loading spinner,
but thereafter we want return to make sure
that none of this code here executes
and we're not running into this error, which we just saw.
Now, with that added, you actually learn though,
if you hover over these blue squiggly lines,
that response body can't possibly be null.
And indeed, if I hover over body, it's always a string.
There is no question mark after string.
So this is never null.
Now, we saw null in the debug console
because actually what Firebase does here
is it returns to string null,
so a string with the text null inside of it.
So this logic here should be backend specific.
Some backends might return an empty string.
Others do something like Firebase
and return null in a string,
yet others might use a certain response status code
for which you want to check, for example,
a 404 or anything like this status code.
So it depends on the backend.

For Firebase, it's this check here,
which you want to add to make sure
that we're getting rid of the loading indicator,
but that we're not going through this logic here
if we got no data from the backend.
And with that, if we save that and reload,
you see that now we only briefly had that loading indicator,
and then we see that fallback text.
So adding this logic here was one thing I wanted to do.

14. Better Error Handling

So handling this no data case here
was one thing I wanted to do.
The other important concept I wanna dive in,
is another form of error handling.
Now of course here we are already handling errors
by checking for the response status code,
but with that we're actually handling error status codes.
So errors returned from the backend through the response.
And this is a very important aspect of working with HTTP,
but it's not the only thing that can go wrong
because sending this request here could for example,
also go wrong because you maybe have no internet connection
or anything like this.
And in such a scenario,
Dart and Flutter would actually throw an error.
They would generate an error object and throw
that error object with a mechanism that's built into dart.
You can throw an error with the `throw` keyword
and then here you can create such an exception object
by using the exception constructor function,
the exception class that's built into dart

and give it a message like an error occurred.

That's how you could manually throw an error.

Now you don't wanna do that too often,
because that will actually crash your application
if you're not handling it properly
though the error handling is the part
into which we'll dive over the next minutes.

But that's how you could throw an error.

And as you see here, all this code now got grayed out
because if you throw an error,
no error code after throwing that error will be executed.

Now here I don't wanna throw my own error,
but it's worth knowing that this feature exists
and that you can generate such an error.

But why would you do that?

Well, because you might wanna communicate that something
serious happened, something that leads to all the other code
that would normally execute thereafter being redundant
and therefore the app should handle such errors
to get back to a state where it works properly again,
and that's what we should do here
when using Git or all these other HTTP related methods here.

Because these methods actually will throw an error
if something goes wrong with sending that request.

Not if you get an error or a status code
that should be handled as we're doing it here,
but you would for example,
get an error if you have no internet connection
or if you're, for example using an invalid domain here.

If I remove the.com in such cases,
this would also throw an error.

And how can you then handle such an error?

Well, that can be done with two other crucial keywords
that are built into dart,
the try keyword and the catch keyword.

Now try takes a pair of curly braces after it,
and you should put the code that could potentially fail
into this try block.

So between those curly braces that belong to try.

Catches then added after the closing curly brace
of the try block,

and catch, as the name suggests,
catches any error that might've been thrown by the code
between those try curly braces.

You then get access to the error here by wrapping
any variable name of your choice between parentheses,
and adding another pair of curly braces opening and closing.

So this error thing here is in the end a variable
that's created automatically by dart,
and you can give it any name of your choice
whenever this code here between those try curly braces
causes an error.

And this is then simply this exceptional object
which I showed you before with throw.

So this exceptional object which you can create,
that's the error you are getting here
when catching such an exception.

And this error object could contain useful information
like for example, a more detailed error message,
and you can then add your fallback code
that should be executed when an error occurs here,
in this catch block.

And thereafter, the app will resume in a normal way.

Therefore, here I will actually wrap all this code

where I transform the data,
load the data, and so on with this try block.
So I'll cut it and add it here instead of try.
So that I try executing all this code,
but if it should fail, we make it here into this catch block
and in there I then also wanna set my error state
as I do it here in case of an error response.
But now I'll do it inside of catch.
And here we can then say,
something went wrong, for example.
And if we do that and we now still have this broken URL,
which, and that's important broke
because here I removed the domain.
If I reload this, you would see this error message
and without try-catch the app would've simply crashed
or would've been put into some undesirable state.
Now instead, we're handling this error gracefully
and that's why you should consider adding error handling
like this with try catch
in addition to handling error responses.
Now, you might also wanna do this when sending a request
in a new items here to keep things
a bit simpler and shorter.
I'll only do it in load items, but being aware of try-catch
and of this way of handling errors is important.
With that, I'll fix this error here and re-add the domain,
but that's how you can use try-catch
for additional error handling.

15. Using the FutureBuilder Widget

Now to conclude this course section,
there's one special widget I want to show you.

A widget you don't have to use,
but a widget that can help with loading data
through HTTP requests,
or with working with futures in general.
And keep in mind that, of course,
when sending an HTTP request, like here with `get`,
we are getting such a future as a result.
And that widget would be the `FutureBuilder` widget.
A widget which listens to a future
and automatically updates the UI as the future resolves.
So, as the future produces some data.
Now this `FutureBuilder` widget could be used here
in the `grocery_list.dart` file in our `build` method.
And they're actually inside of our scaffold,
in this place here where we have the content.
Here, instead of outputting the content,
we could use the `FutureBuilder` widget.
Now, the `FutureBuilder` widget
needs at least two key parameters.
It needs a future.
So, the future to which it should listen in the end.
And a builder which wants a function that will be executed
whenever the future produces data.
And therefore that is a function
that receives a context automatically passed in by Flutter,
and a snapshot, which in the end
gives you access to the current state of the future
and the data that might have been produced.
Both arguments are passed in by Flutter
because Flutter will call this function
based on the state of your future,
and therefore, now here, we also need a future.

And here, for this future in this scaffold widget
in my `grocery_list.dart` file,
I in the end wanna have a future that produces that list
of grocery items once we got a response
from this get HTTP request.

Now, to get such a future that produces that data,
we could change the `loadItems` function.

Instead of returning void here,
we could say that this function should return a list
of grocery items.

Now here, we'll then actually get an error,
because since we're using `async` here,
this will automatically return a future.

As you see here.

This `async` keyword basically ensures that the function
that's decorated with it will always produce a future,
automatically, Dart will take care about that.

Therefore, the correct type annotation
for this function here, for this method,
is that we'll get back a future
which will eventually resolve to a list of grocery items,
which can be expressed with this generic type.

So now, `loadItems` should be a method that returns a future,
which it will do automatically
because of `async` as you learned,
which will then in turn resolve to a list of grocery items.

To make sure that that's the case,
we have to go to the place here at the end
where we set our state
and get rid of this set state action here,
and instead return our `loadedItems`.

So, we return that list of items.

I will also get rid of this catch block,
and of this curly brace that belongs to the try block,
because I will get rid of this overall try catch block
because with FutureBuilder, as you will see,
we can handle errors in a different way,
in a way that requires less code from our site,
which can be quite convenient.

Now here, in this case here, where we got no response,
I also don't wanna return nothing.

Instead here, I simply wanna return an empty list.

because we got no response, hence we got no items.

Therefore, I wanna return an empty list.

Because, keep in mind,

loadItems should now return a list in the end,
a list of grocery items.

Now we also don't need to set the loading state
manually anymore, thanks to FutureBuilder
we'll soon do that in a different way.

And here where I'm setting this error state,

I'll comment this out

because we'll also soon handle this in a different way.

So, that with all those changes made,

we got a leaner loadItems function,

which still sends this HTTP request

and transforms all that data,

but which then in the end returns that transformed data.

Now, down there in the build method, at the very end here,
it's now, therefore, my loadItems function.

Or to be precise, the result of calling that function
that will give me the future that should be assigned here.

But this code here is actually not correct.

Technically it would work,

but it's discouraged to write this code,
and it's considered a bad practice to write this code.
Because if you call `loadItems` here to get that future
that will be passed as a value to `FutureBuilder`,
this function will be re-executed
every time the `build` method executes again.
So, every time some state changes, for example.
And we don't want that.
We don't want to re-execute this all the time.
Instead, this function here should only be executed once
when this widget, this grocery list widget here,
is loaded for the first time.
For example, because the app started up.
That's the only time where this should be executed,
because that's the only time
where we wanna fetch all those items from the backend.
Therefore, instead of calling `items` here,
you should instead add the `initState` method,
which you can add here,
or which we already did add here to be precise,
to actually create your `loadItems` future there.
Now for that,
I'll actually get rid of this `isLoading` variable
because we won't need it anymore.
And instead I'll add a new variable,
which will be of type `Future`,
which then eventually resolves to a list of grocery items,
and I'll name it `loadedItems`.
And initially I have no value here,
but I will store a value here in `initState`.
There, I'll set `loadedItems` = to calling `loadItems`.
`loadItems` then will produce that future.

And that's therefore exactly what I store in loadedItems.

Now here I'm getting an error

because I got no initial value here,

and this is a non nullable value.

And to work around that, we can use a special keyword,
the late keyword, which we can add in front of this type.

This is a keyword built into Dart, which in the end,
tells Dart that we have no initial value for loadedItems,
but we will have a value
before it's being used the first time.

And indeed this is the case here,
because we do assign a value here in initState,
which will be executed
before the build method executes in the end,
and before this future is needed.

So with late, we basically tell Dart
that everything will be okay.

So, then we have the loadedItems property,
which holds a future,
and it's now that property that we can use down there
as a value for future on that FutureBuilder.

And this will not only be set once when initState runs,
but it's then this future to which Flutter will listen,
and for which it will execute this builder method,
as this future changes its state
from being initialized and pending, to being resolved
or rejected in case an error occurred.

And now speaking of these different states,
now that we got this future in this builder method,
we should return a widget.

And typically we don't just wanna return one widget here,
but different widgets

based on the current state of the future.

And for that, we got this snapshot here.

We can then add a if check here in this builder function,

for example, and take a look at this snapshot,

and then use this snapshot to evaluate

the current state of that future.

And for that we can use the `connectionState` property

on this snapshot object,

which can have a couple of possible values.

For this, we can use the `connectionState` enum

that's built into Flutter.

And here we got the different values

`connectionState` could have.

For example, it could have `waiting`,

and that will be the initial state we sent the request

and we're still waiting for the response.

So, that's our loading state in the end.

And if we are waiting for a result,

we might wanna return our loading spinner.

So I'll grab that code here,

where I render this center widget

with the circular progress indicator,

and I'll cut it and get rid of this if statement here.

And instead, down here, I wanna return this

if our current `connectionState` is `waiting`.

Now if we're not waiting, that means that we have a result.

But the result could now either be that we have data,

or that we have an error.

So therefore, here we want to add another if check,

and we can use this snapshot object again

to check whether we might have an error

with this built-in `hasError` property.

This will be true if our future has been rejected,
which simply means that an error has been thrown
inside of the code that produces the future.
So in `loadItems`, if we throw an exception,
the future will be rejected.
For example, if `get` throws an exception
because of a missing internet connection,
the future will be rejected.
And here where we're manually checking
for an error status code,
and we previously set the error property to a message,
we can now throw an exception ourselves.
And that's a great example for a scenario
where you do want to throw your own exception.
Here, we're doing it to make sure that this future,
which is produced by `async` automatically, will be rejected,
and therefore, down here in our builder,
`hasError` will be true.
That's why we, for example here, wanna throw an exception.
Now we can pass our own error message here
to this exception, like `Failed to fetch grocery items.`
`Please try again later.`
And with that, the future will be rejected.
And now, down there in our builder method,
in this `if` check where we're checking for the existence
of an error, I now wanna return a different widget
if we make it in there.
And that widget will be this center widget,
which we used for displaying an error in the past as well.
And we can get rid of this `if` check here, therefore.
And it's now this center widget which I wanna output here,
but I'm not referring to this error property now.

But instead, here, I now want to use snapshot.

And then, there, I got this error property,
which gives me access to this exception object in the end.

And here, I can call toString.

And that should normally output the error message.

Now, of course, we also want to handle the case
that we're not waiting and that we don't have an error.

So, put in our words, that we have data.

And for that, here, I wanna output my list,
or this fallback text if I have an empty list.

So, therefore, I want to add yet another if check here
in this builder method,
where I can check if snapshot data,
which is the data we will have in the end,
if that isEmpty.

Or if we wanna show the fallback here, if it is empty.

So, if we have an empty list here.

Now data will be such a list of grocery items, or null.

Though here we know that it won't be null
because we don't have an error
and we're not waiting for the future to resolve.

So, at this point, we should have data,
and therefore, here I add this exclamation mark
to make it clear that here in this app this won't be null.

But if it's empty, if it's an empty list,
then I of course wanna return this fallback text.

So, I'll grab this center widget with this fallback text,
get rid of that variable here, and now show this here.

And if we also make it past this if check,
we know that data is not empty,
that it is a list of grocery items.

So, now we can grab this list view here, cut that,

get rid of this if checkup here in the build method,
and now down here at the very end of this builder method,
and then here we add our ListView.builder.

So, this ListView widget.

And there, we're now outputting all these _groceryItems.

However, here, we should not use this property,
because here, where inside of this builder function,
it belongs to FutureBuilder.

And as mentioned before, this builder function,
in the end, exposes the state and the data produced
by that future through this snapshot object.

Now this loadedItems future, in the end,
of course, is the future produced by loadItems,
and that future either produces an error
if we throw an exception,
or if any other code in there throws an exception.
In that case, as mentioned, the future will be rejected,
and that would lead to this hasError property
on snapshot yielding true, and this widget being returned.

But if it's not producing an error,
loadItems will produce a list.

It will resolve to a list.

The loadedItems list, or an empty list.

And that list is available
through that snapshot object as well.

So, in here, in that builder function
that belongs to the FutureBuilder,
inside of this function here, we want to use this snapshot
to get hold of the loaded data through snapshot.data,
and then here an exclamation mark to tell Dart
that this will not be null, which we know
because if we reach this point,

we will have some data in this app here.

So, in all the places where we use `groceryItems`,
we wanna use snapshot data with that exclamation mark
at the end here instead.

With that, we should be outputting the loaded data.

And if you now save everything and reload this app,
you should see any data you have on the backend here
being loaded.

So, we have the loading spinner initially, briefly,
and then the data.

But you will also see that for this app,
we also got some problems.

For example, add a new item like milk here;
you will see that it's not showing up here.

Only if I restart the app, it is being loaded.

So, it was sent to the backend,
but somehow the UI was not updated correctly.

If I delete an item, we get an error.

So, that's also not good.

And the reason for all these problems
is that now we're loading and displaying our list data
through `FutureBuilder`, but in the end,
this builder function here only executes once
when this future is created and executed for the first time,
so to say, once the future is done.

So, once it resolved with data,
this builder method here will never execute again.

Even if you call `setState`
and the `build` method executes again.

That will recreate the `FutureBuilder` but not the future,
since that future is created in `initState`,
and therefore only once, and therefore the builder method

of FutureBuilder also doesn't execute again.

That's why this UI never updates,
and why this list, for example, never updates.

That's why for this app here,
where we're loading data that's then actually manipulated
from inside the same widget,
FutureBuilder might not be ideal.

Now you could restructure the code
and write it in different ways to still make addItem
and removeItem work as before.

But for this app, it actually might be preferable
to simply not use FutureBuilder.

Nonetheless, I of course wanted to introduce this widget
because it is an important widget.

You will also see it later in the course,
in a place where it's more useful,
and you might also see it in other projects.

In addition, you can, of course, hopefully,
see that this is a very elegant way
of handling these different states a future might have,
and of outputting different content on the screen,
even if in this app it's not ideal
because of the complexity of this screen.

But if you had a screen or a widget
where you only need to load data,
show different states based on whether you are downloading
or not, and you have no other logic related to the data,
FutureBuilder might be ideal.

So for this app here,

I would actually go back to the state we had before,
and I just wanted to introduce you to this widget.

But in our projects, FutureBuilder could be very useful

because it can simplify this process of rendering different UI states based on the current state of a future.

16. Module Summary

And with that,

that's it for this course section.

Now you learned how you can send HTTP requests, for example, to post data to a backend to send a post request, which is typically used for storing data on a backend, though it will, as mentioned, always depend on your backend, which kind of request is needed.

You learned how you can fetch data, and also how you can show a loading spinner or possibly an error message if needed by managing appropriate variables and updating the state as needed, and then outputting conditional content.

And you therefore also learned how to check for possible errors, and of course, you also learned how to extract the data that's sent back, though that will of course also depend on your backend, how the data is formatted and with which data you have to work will depend on your backend.

Here in Firebase case, we're decoding that JSON response body and we're then transforming it such that in the end, we again have a list of grocery items.

With all these things, we're sending data back and forth, and now we connected this app here to a backend

so that the data would not be lost
if the app would be deleted
or the device would be replaced,
but that instead, the data is stored on a backend server.
Now, in this case here, it's a very simple dummy backend,
and indeed here, all users would be using the same database
and would therefore see the same list,
but that's just a demo.

Of course, for more advanced use cases,
you could also add user authentication
and protect the data.

And if you would want to do this with Firebase,
the Firebase documentation also lets you know
how you can authenticate your requests,
and also how you could scope data to users
so that not all users can see all the data,
but that's something that's related to the backend,
not to the frontend here.

This section was about getting you started
with HTTP requests
so that you can build apps
that do not just store data locally,
but also remotely, on a database typically,
with the help of a backend.

12_fav-place-app_using-native-device-feature

1. Module Introduction

This course is
about building native mobile apps
for both iOS and Android with Flutter.
But up to this point,
we haven't really used any native device features
like the camera or getting the user's location,
and that's therefore exactly
what this course section is all about.
Because now that you have all these crucial Dart
and Flutter features,
and you already know a lot about building mobile apps
with Flutter and Dart,
it's time to dive into these native device features.
And for that, in this course section, you will learn
how you can access the device camera
to let the users take pictures and images
and how to then use these images in your app.
You will also learn how to get the user's location
and how to display that location on a map using Google Maps.
And you will also learn
how a user could alternatively select a location on a map.
And we will actually dive into many more map-
and location-related features.
And you will also learn
how you can store data on the device,
because thus far, we only stored data in memory
where it would be lost if the app was running

in the background for too long, or if the app were closed,
or we stored it on a backend,
which are perfectly fine use cases.
But sometimes you also have data
that should be stored on the user's device.
And that's exactly what we're going
to dive into in this course section.
Well, and as always, along the way, you can practice
what you learned thus far,
and we'll dive into some other new features and concepts.

2. Setup & A Challenge For You!

Now, for this course section here,
I prepared another new starting project,
a new main.dart file, which you find attached,
and which you should download
and use to replace your main.dart file
in a new Flutter project.
So here, I created a new project called FAVORITE_PLACES,
and in that project,
I replaced my main.dart file with the attached one.
Now, if you take a look at the content
of the attached main.dart file,
you will see that in there we seem to be using Google Fonts.
I'm using it to set up some text themes
based on Google Fonts here.
That's the reason why I'm adding Google Fonts.
And this is how you can use Google Fonts
to set up text themes.
In addition, I'm setting up some base colors here
based on a seed color,
overriding one of the colors

that would be inferred automatically alternatively
with a more specific color.

And then if you scroll down,
you'll find a MyApp widget, a StatelessWidget,
which simply sets up the theme,
a title for the app, and then nothing more
because we'll write the other code together.

Now, before we write any code though,
you should make sure
that the Google Fonts package is available.

And for this, you should run
`flutter pub add google_fonts`
to add this package to your project
so that this import, then, can be resolved.

And therefore, ultimately, after running this command,
you should have a new entry in your pubspec.yaml file
where you do find this google_fonts package
under dependencies.

The version, of course, might be different for you,
but the package should be there.

And with all of that out of the way,
we are ready to dive into this project.

And now, before diving into this project,
I got a challenge for you.

I want you to practice
some of the key concepts you learned up to this point
because practice, of course, is important.

And therefore, I want you to build
this very basic first version of the app,
which does not yet use any native device features
because we're going to dive into that together later.

But this is an app which actually supports multiple screens

and navigation between these different screens,
and it should actually also use Riverpod,
that state management library
about which you learned earlier
for managing some data.
To be precise, for managing some favorite places.
Now, in this basic first version,
a favorite place
will actually just be an object with a title.
Later, we'll also add images and locations to places,
but we're going to do that together.
Therefore, here in this first version,
a place should simply have a title
and actually, also a unique ID.
So, these two pieces of data
should be stored in a favorite place,
and the favorite places should be managed with Riverpod,
so that a place is added on this add place screen
with River Pod in the end.
And then, this list of places or the place detail screen
where you can view the details of a place.
These screens should get their data with help of Riverpod.
Now, this app does not use any backend,
no HTTP requests are needed.
Instead, your challenge simply is to build these screens,
set up a place model,
and use Riverpod for managing the data.
Definitely try this on your own first.
It's a great exercise.
We're of course going to implement it together
over the next lectures though,
before we then thereafter

dive into all these native device features.

3. Adding a Place Model (Challenge Solution 1/6)

So, did you succeed?

Let's now build this together,

and I am going to start

by setting up a model in a models folder.

In there, I'll add a new place.dart file

in which I want to define my model,

my blueprint for my place objects.

For that, I'll create a new class named Place.

Of course, you can choose a different name,

but, for me here, Place makes sense.

And then here, as described in the previous lecture,

every Place should have an id,

and every Place should also have a title.

Now we also need a constructor,

and I mentioned that the ID should be autogenerated.

Therefore, here I will only add the title

as a named parameter,

but this will actually be required then.

But for the id, I'll instead use such an initializer list

about which you learned earlier in the course,

and I'll use an extra package for managing that id.

And the extra package I want to use

will be the uuid library,

which can be installed

as described in their installation instructions.

So here I'm going to run this command to add the library,

and then it can be used as described in their Readme

which, for me here, means that in the end

I want to create such a uuid object,

and then we can call the v4 method
to generate an id that looks like this.
That is totally random as you can see.
Alternatively, you could also use one of the other methods
to, for example, create a time-based unique id,
but this is up to you.
So here in this Place model file,
I'll import the uuid package,
and then, from there, the uuid.dart file,
and I'll create my new uuid variable
or final variable here
by instantiating uuid like this.
And actually, since this is const,
I can create a const data container instead of a final one.
And I'll then use this to assign my id to an initial value
of uuid.v4 like this.
Then this can't be const anymore though
because now I'm deriving a value dynamically here,
and that does not work with a const constructor.
But this is now the finished Place model.

4. Adding a "Places" Screen (Challenge Solution 2/6)

With that finished,
it's now time to add some widgets and some screens,
and here I'll add a screens folder
to hold my screen widgets,
and a separate widgets folder
which will then later hold any widgets
that are not standalone screens.
But I'll start by adding a screen here.
To be precise, I'll add a couple of screen files already
and then we'll, step by step,

add the widget classes to those files.

For example, we'll need a `places_list.dart` file

for this, well, list screen

that shows all the places,

a `places_detail.dart` file

for showing the details of a place.

And then also an `add_place.dart` file,

which will display the screen we use

for adding a new screen.

Now here I'll start with the `places_list.dart` file.

In there, we create a new class,

and I'll name that class `"PlacesListScreen."`

Here, again, working with `"Screen"`

being added to the end of that name,

which is of course optional,

but which is something I am doing here

to make it really clear

that this widget will be loaded as a screen,

which the widgets in the `widgets` folder later won't be.

And then here I want to extend `state full`

or `stateless widget`,

and for the moment, it will be `stateless widget`.

Of course, we also must add that import to `material.dart`.

Now, as always,

we should add a constructor `PlacesListScreen`,

where we, in the end, accept and forward our key.

And right now, no other parameters would be needed here.

And we, of course, add the `build` method

where we wanna return our screen widget tree.

I'll start by adding a scaffold

because, of course, I want to have that base screen set up.

I, for example, also do want to add an app bar here

by using the built-in AppBar widget.

And I'll use that app bar to set a title.

That will be a text widget,

which here could simply be "Your Places,"

because this screen is all about displaying the places

that were chosen by a user.

Now besides the title,

the app bar should also display an icon button

that can be pressed to navigate to the "add place" screen.

For that we can add actions here, this list of actions.

And in those actions, we can add an IconButton,

as we did it many times before in the course,

of course.

Now, this IconButton needs an icon.

That's one of the parameters you must set.

And here I'll create an icon with built-in Icon widget.

And the icon I want to use is the good old "add" icon.

Now, besides Icon,

we, of course, also must set the onPressed parameter

to a function that will be executed

when this button is pressed.

Right now I don't have the logic for that yet,

but that will change later.

So that's the app bar.

And with the app bar added,

it's, of course, now also time

to add the body to this scaffold.

So the main page content.

Now for displaying that list,

I'll add a new widget in the widgets folder

and we could name this file places_list.dart as well.

That would be fine, since it's in a different folder.

But since I don't wanna have a clash in file names here
and cause any confusion,
I'll keep places_list here.

But I'll rename my screen here to just "places."

I just wanted to highlight
that you could have the same file names.

That would be no problem,
but to not confuse you
when I say that I'm going to places_list
and you then wonder which of the two files I mean,
I will choose different names here.

So with that done,
I'll also rename my widget here in the places.dart file
from "PlacesListScreen" to just "PlacesScreen."

And in this places_list.dart file,
I'll add my PlacesList widget, not PlacesListScreen,
but just PlacesList,
because this won't act as a screen.

And here I'll extend the stateless widget
and add the appropriate import.

And then in here, the idea is to have a list of places
using my Place model
and adding the appropriate import for that as well,
of course.

And this can be named "places"
and that list should be set with help of the constructor
so that the places are passed into this widget
where I now don't just accept and forward the key,
but where I also require and set the places.

Well, and with that, of course,
we also must add this build method, as always.
And then in the build method, I'll return my list of places,

or to be precise,

I wanna return either my list of places

or if I got an empty list, I wanna return some text

that tells the user that there are no places yet.

So in the build method here,

we can simply check if "places" is empty.

So if we have no places,

in which case I wanna return,

let's say, the center widget with "const" in front of it

with a child of this text here using the text widget,

where I simply say "No places added yet."

Quite straightforward and simple.

Alternatively, if I make it past this "if" check here,

I know that I do have some places,

and in that case, I wanna return a list view

and I'm using list view here to get this scrollable list.

But also because I want to use

this builder constructor function here,

because this could potentially be

a very long list of places.

And of course, everyone, I have great performance

and make sure that only those list items are rendered

that currently should be displayed.

Therefore, we need list view builder here.

And then we add our builder function

where we get the context and the index

and then return the list item that should be displayed.

And of course, we'll also need to set the item count

and I'll set the item count here to places.length

to give Flutter an idea

of how many places will need to be rendered in the end.

Well, and then for every place here,

I, in the end, want to render a list tile widget,
which is one of those widgets
you learned about earlier in the course,
which is simply a widget that's optimized
for being displayed in a list.

And in this list tile here,
I will simply set the title parameter to my place title.
To be precise, to a text widget that outputs my place title.

So here, to get that title,
we can use "places,"
acts as a single place by index.

And then since "places" is a list of place objects,
we can access .title
to output the place title here like this.

Now I'll just also add some styling to this text here
by reaching out to my theme with "Theme.of(context)"
and then there, I want to use the text theme.

And then there, specifically,
the title Medium theme object or Data Object,
which will be set,
which I'm telling Dart by this exclamation mark,
which is an annotation I must make here
to then be able to call "copy with" on that text tile
so that I can overwrite some selected properties
of that text style in this place here only.

So for this text widget here only.

And the thing I wanna overwrite here
is the color of the text,
which here should be set to Theme.of, again,
reaching out to the theme color scheme.

And then they are the on background color.

And of course, if you set up that styling differently here,

that is also absolutely fine.

But with that, I have that list view builder,
which outputs a bunch of titles.

That's all it does at the moment.

And we can now use this places list widget
on which we worked in the places.dart file,
where it should be output as a body.

So here I wanna output PlacesList like this,
and at the moment I'll set places to an empty list.

This will change soon, but right now it's empty.

I'll ignore the blue, squiggly lines
because as mentioned, this will change soon anyways.

And now I can also use this PlacesScreen here
in the places.dart file.

I can use that in the main Dart file
to load it as my home widget.

So here, I'll display PlacesScreen,
and that can be set to const.

With all of that, I can run this app here on the emulator
and you should now this fallback text here.

Now, of course,
this fallback text is a little bit hard to read,
so to make it a bit more readable,
back here in the places_list Dart file,
we should also apply some styling to that fallback text.

And I'll pick this same styling as I used here
for my text widget inside of the list tile
and apply this here.

Now, this just can't be const anymore,
because this is now dynamic
and I will actually use "bodyLarge" here
instead of titleMedium so that I got this text.

But with that, that's looking all right.

Of course, you could add more to this fallback screen,
an image or anything like this,
and of course, always feel free to do such things
as an exercise.

But I'm happy with that.

And therefore, I now wanna focus on this plus button
and make sure that it loads an "Add new place" screen,
which then also allows me to add new places.
And then, of course, we'll also add River Pot
to manage those places and so on.

5. Adding an "Add Place" Screen (Challenge Solution 3/6)

So let's now work on this `add_place`
screen file here, on this `add_place.dart` file.

In there of course, as always we add a new class.

`AddPlaceScreen` sounds like a fitting name

where I'll extend `StatefulWidget` here,

`StatefulWidget` because I will manage

some user input here and therefore here,

I of course then also add my constructor

function where I accept and forward my key.

And we add this `createState` method here,

which returns a state that's connected

to a `AddPlaceScreen`.

And of course therefore we also must add this

`_AddPlaceScreenState` class, which extends `state`

and `state` is a generic type as you learned,

connected to `AddPlaceScreen` here.

And then in there we add the `build` method.

And now in `createState` we can return a new instance
of `AddPlaceScreenState` like this.

In this build method,
we then wanna return our input elements in the end
that allow the user to input the data for a new place.
And I'll start by adding a scaffold
because this is a new standalone screen.
So of course I wanna do all that basic screen setup.
For example, add an appBar as pretty much always
where I'll set a title that is add new place
wrapped in a text widget, though.
So here we have Add new Place like this.
And of course then we also have our body
and we could outsource this body into a new widget,
a new standalone widget if we wanted to.
But here I'll instead display a column
and handle it all in this one single file,
which I will actually wrap with a SingleChildScrollView
to make sure that it's always scrollable
and all parts of that column can always be reached
no matter what your device size or orientation may be.
So therefore I'll wrap this in SingleChildScrollView,
but then in this column I wanna have the different inputs.
And here we could wrap this all into a form
and use these form features about which
you also learned earlier in the course.
And if you did that, that's of course absolutely fine.
Here I'll keep things a bit simpler
and I'll therefore just use the regular TextField
for rendering a text input element
that allows the user to enter a title.
Now to tell the user that they should enter a title here,
I'll add my decoration parameter
and set this to an input decoration object using Flutter's

InputDecoration class so that I can set my label
to a text widget that says title.

By the way, alternatively,
there also is a labelText parameter,
which you could set which directly allows you
to set your label text without having to add
that extra text widget.

Both is possible.

And since I haven't shown you this parameter before,
I will do it here.

So with that, I set a label on this TextField.

In addition, I wanna add a controller
to let Flutter manage the user input for me.

Now you learned about controllers earlier
and if you did not use a controller,
but you maybe instead manually managed the input
by listening to onChanged and updating some variable
whenever this event triggers, that's also fine,

but here I'll use such a controller

and in AddPlaceScreenState, I will
therefore add a new _titleController,
which is a text editing controller.

And as you learned, if you do use such a controller,
you should also dispose of it.

So I will add dispose here
and access my titleController to then call dispose on it.

And with that added, I can use the titleController here
as a value for controller on this TextField to make sure
that the user input is managed by Flutter for me
and I can conveniently extract it when I need it later.

And this here can be const now, this InputDecoration,
and with that I got this title TextField all set up.

Now later we'll add more inputs to this screen here.

We'll allow the user to pick an image

and to choose a location.

But for the moment, a title is all I need.

And therefore with that added, the only other thing

I wanna add to this column is a `ElevatedButton`

that can be used to save that input to submit it.

I will also add a `SizeBox` between the `TextField`

and the `ElevatedButton` to add some vertical spacing

by setting the height here to let's say, 16.

Of course you can play around with that value, though.

And then here on the `ElevatedButton`, I'll set the child

to a text widget where I say add place.

And this can be set to a const widget by adding `const`.

And we must add the function as a value for `onPressed`

that should be executed when that button is pressed.

So that's also something I'll add here

and some commas for better formatting.

And I will actually even tweak that `ElevatedButton`

and convert it to an icon button so that then I have

to set the label to control which text will be displayed.

And I have to add an icon parameter

so that I can also show an icon in addition to this text.

So here for that I create a new icon

by using the `add icon` again.

Now doing so is of course totally optional,

but that's the style or the look I'd like to have here.

Now one additional thing I wanna do

is I wanna add some padding around my column.

And the column itself doesn't have a padding.

It also has no margin parameter that I could set.

Now we could wrap column in a padding widget

and that would be absolutely fine,
but conveniently, if you take a look at the parameters
provided by `SingleChildScrollView`,
you can see that there we can also set a padding.
So since I used `SingleChildScrollView` here,
which of course you might not have done,
which is also fine, but since I used it here,
I can use that to add a padding
by setting `const EdgeInsets`
and then adding a padding of 12 in all directions.
And that's then this finished `AddPlaceScreen`
at least for the moment.
And of course now I wanna take a look at it
and be able to navigate to it
and navigating to it should be possible
from inside the places screen
because that's where I got this icon button
in my actions list in my `AppBar`.
So here when we press this, I wanna navigate
to this add place screen.
And now you can add your navigation logic here in this
anonymous function or in a standalone method.
And here, since I will also need to use context
and it will be readily available in this build method,
I'll go for this anonymous function approach
instead of adding a standalone method, which would then need
to accept the context as an argument.
Therefore I'll do it right in here in this anonymous
function that's assigned to `onPressed`
and access the builtin navigator class
and created with `of(context)` so that we can then access
`push` and `push a new screen onto the stack of screens`.

And as you learned, you typically push a `MaterialPageRoute` then which has that builder where you get a context and then return the screen, or the widget to be precise, that should be loaded as a screen.

And here I'll of course return `const AddPlacesScreen` like this.

That's my navigation logic here.

And with all of that added, we should be able to save this, maybe force a reload.

And if I now click on plus,

I'm taken to this add new place screen,

which doesn't have too much content,

but looks good for now, which has this add place button,

which of course doesn't do anything just yet,

but where I can also go back and forth and that all works.

And therefore now the next step is to make sure that as we enter something here, I also saved that.

And now that I did enter something here,

I see one last adjustment we should make

because that's pretty hard to read here.

We should go to this `add_place` screen,

and in there, make sure that this text

in this text field is always readable.

This can be achieved by setting the style parameter here,

which allows us to set a style

for the text that is being edited.

And here I could again use theme of context to use one of the built in text styles,

but I'll instead create a new text style

just by creating a new text style object

where I set the color to `Theme.of(context).colorScheme,`

and then there the `onBackground` color,

which should be some whiteish color.

With that, this looks much better.

So now I got this, and now I wanna make sure
that pressing add place also creates a new place
that is then shown on that list.

And for that, as mentioned, we will use Riverpod.

6. Adding "riverpod" & A Provider (Challenge Solution 4/6)

So it's a River Pod time.

Because I wanna manage my places in some app wide state
that can be accessed by different widgets
from any place in the app.

For that, I will have to install
this Flutter River Pod package
about which you learned earlier.

And this can be done with `flutter pub add flutter_riverpod`.

Of course, you can also simply search for flutter riverpod
and visit the official documentation page
or the package page to learn more about how to
use this package and how to install it.

And with it installed,

we can go back to our project here.

And in there I'll add a new folder, in the lib folder,
which I'll name providers

because you might recall that with riverpod
you can set up so-called state providers.

And in this providers folder I'll add one new file,
which I'll name `user_places.dart`

which will be responsible for managing those places
selected by the user or added by the user.

Now in this file you should add a import
to this Flutter River Pod package

so that we can use its functionality in this file.

And then here I want to add a notifier class

because you might recall that you can create

such notifier classes with River Pod

to have a class where you can add methods that allow you

to manipulate the state that's managed by River Pod

and that's exactly what I wanna do here.

So I'll add a user places notifier class here,

which must extend state notifier,

which is a class provided by a River Pod.

And state notifier is actually a generic class

where we should add some annotation that tells

River Pod and Dart which kind of data will in the end

be managed by our notifier class here and here.

And here the data I do wanna manage

will be that list of places using my custom place model.

And for that of course we must import that model

by reaching out to our own package,

which in my case here is called favorite places

and then there to the model's folder and then place start.

So that's the base setup.

In addition, you then must add a constructor function,

which doesn't take any parameters,

but which must have an initializer list

so that we can reach out to the super class

by calling Super, so to the parent class,

and we can pass on our initial state.

And the initial state here will be an empty list

and it's actually marked as const

because as you might recall the state managed

by a River Pod must not be mutated.

You must not edit it in memory.

Instead, if you want to add something to it,
you should create a new state object,
a new list in this case here,
that could be based on the old state,
but you should not edit the old state directly.
And we make sure that we don't accidentally do that
and we instead would see an error if we tried to do it
by adding `const` in front of that list.

So with that here I'm setting up this initial state
and now we can add a method here to this class
that allows us to add a new place.

For this I'll add a new `add place` method
where I expect to get some input.

To be precise I want to get my title of the place
that should be added and then in there
the goal is to create a new place object
and then that place object should be added to this list.

By the way, if instead of the title you came up with
a solution where you expected the finished place
as an input that would of course also be fine.

And in general,
of course your solution will deviate from mine.

It's all about practicing though,
so by taking a look at my solution,
you might get some new ideas
or you might also have some places
where you prefer your solution
and that's also absolutely fine.

Here I'll accept a title though
and I'll then create a new place
by instantiating my place class
and setting the title parameter

to the title I'm getting here.

With that new place added we want to add it to our state.

And as mentioned, we're not allowed

to directly add that state.

So therefore I'll instead set this state property

that's provided by state notifier equal to a new list.

And in that new list I'll use that spread operator

about which you already learned

to spread my old state elements into this new list

so that we don't lose them,

but in addition I'll add my new place here like this

so that it's always at the start of the list,

but that's of course optional.

With that we have a method that will update the state

and that's all we need for now.

We now just also need to set up a provider,

a user places provider, for example,

by using the built-in state notifier provider class,

which is created like this,

which takes a function as an input

to its constructive function.

And in this function you automatically get

such a ref object which you could use to listen

to other providers as you learned.

But here, I'll instead just return a new instance

of my user places notify class like this.

And that is all, it's a very simple provider

and right now also a very simple class.

With that provider added here,

we now have to make it available to the entire app,

which can be done from inside the main dart file

where I now want to import something.

For example, I want to import
from the Flutter River Pod package,
the Flutter River Pod dot Dart file.
And I'm just organizing my imports such
that the package imports come first
and then come the custom imports.
And now with that import added here
we can go down to Run app and wrap my app
with the provider scope widget,
which is provided by River Pod.
And my app is now simply the child widget
that's passed to Provider Scope.
And as you might recall,
Provider Scope simply makes River Pod
and the providers we created available inside of our app.
So this is needed to be able to access River Pod
and its features and our providers
from inside off the widgets that make up our app.
So with that added here,
we can save the main dot dart file
and now go back to the add play dot dart file
because there I now want to use River Pod
to add a new place.

7. Adding Places with Provider & Displaying Places (Challenge Solution 5/6)

I'll add a new method maybe
above this pose though the order does not matter,
which I will call savePlace
because that's what I will do in here.
And the idea here is to extract the text that was entered,
maybe also add some validation
and then add that place with help of riverpod.

For that I'll get my enteredText here
by using the title controller and accessing text.
And then I'll check if enteredText is empty.
And I'll actually also check if enteredText is equal
to null or if it's empty
to have these two explicit checks here.
And if that should be the case, I'll just return so
that no other code thereafter is executed.
You could of course also show an alert dialogue
so you could again show a dialogue as you learn that
before in the course.
You could also show some onscreen validation
by conditionally adding a widget
with an error message if something goes wrong somewhere here
in your widget tree.
But I'll keep things simple here
and just do nothing if some invalid text was entered.
And actually here Dart tells me
that this can't ever be null.
So this check basically is redundant and it can't be null
because titleController ensures
that text always yields some text,
even if it's just some empty string.
So with that we have this basic validation.
Now as a next step, I want to make sure that I can save
that place in my provider.
And to do that we need access to riverpod
and to our providers, which as you might recall, we can get
by extending ConsumerStatefulWidget here
instead of StatefulWidget.
And then ConsumerState here instead of state and
therefore create state also will return a consumer state.

And if we do that, we'll have a
ref property available which is
provided by ConsumerState which we can use to reach out
to our providers.

And here in this function
that is triggered when our button is pressed,
we can use the read method on ref to get access to one
of our providers without setting up a watcher.

And the provider to which I do wanna get access here
of course, is my user places provider.

And to get access to it here, you also must import it.

So you must add this import to user_places.dart.

With that we get access to the provider.

And with that provider being used here,
we can then also access our notifier clause that's connected
to this provider here
because it's then this class that can be used
to add a place.

So to call this addPlace method.

And this then allows me to pass on the enteredText
as a title and
therefore maybe we should rename this to enteredTitle
to be a bit clearer about what's stored in this variable.

But with that we'll add it as a new place.

And I then of course also wanna leave this screen,
hence I'll then also use the navigator object again
with of context.

And context is available here in this state class
or this ConsumerState class here actually.

And then as you learned, we can call pop
to leave this screen.

Now with that we should be able to add new places.

Of course, I then also want to be able to show them.

And for that we can go back to the places start file.

So to the PlacesScreen widget.

And in there I want to load my places
and display them.

Alternatively, we could also go to PlacesList
and set up our loader there.

But I'll actually do it here in PlacesScreen, also
because I will adjust my logic a little bit later
and I wanna do that in that screen later.

So therefore in here I wanna get access to my providers
and set up a listener that tells me
whenever my list of places changes.

And to do that, instead of extending StatelessWidget,
we should extend ConsumerWidget,
which is basically the StatelessWidget
provided by riverpod.

For that of course we should also import
the riverpod import.

And then with that we will not just get the build context
here in the build method,
but also an additional second argument which will be
of type widget ref, which is a type and a class
provided by riverpod, which I'll name ref here.

And that's the reference we can use to set up a watcher
by using the watch method, so a listener to data changes.

And we can watch our user places provider.

And of course for that you also must add a import to
that user_places.dart file.

With that, we get access to our userPlacesProvider and we
therefore get access to our userPlaces,
so to the places added by the user.

Now if you hover over that, you will see that at the moment Dart only knows that this will be some kind of object or maybe also null.

To give Dart a better idea of what kind of data will be returned here, we should go back to our provider and to the StateNotifierProvider we're setting up here and take advantage of its generic nature by adding some extra type annotations here, you could say.

By adding these angle brackets and making Dart aware of the fact that this StateNotifierProvider will use the UserPlacesNotifier, but as a second generic type, it will then actually yield a list of places because that is the state managed by UserPlacesNotifier.

And with this annotation added here, as we also did it earlier in the course when we learned about riverpod, dart now knows that this here will be a list of places.

Well and therefore this list of places can now be used as a value here for places in our places list widget.

And with all that added, the last thing that's left is to go to add place

and make sure that this savePlace method on which we worked before is triggered when we click this button.

So therefore we should use it as a value for this elevated button for its on pressed parameter.

And with all that done, if you save this and reload, you should be able to visit this add a new place screen, enter your title,

and click add place.

And it should take you back to your places screen

and you should see your place here.

And if I enter a second place, I can also see

that added in front of the other place

because of the logic we added

to our add place method in the provider.

And of course, right now, this doesn't look too fancy,

but it is a start.

And the only remaining part of the challenge I gave you,

therefore now is to make these items tapable so

that we then go to this place's detail screen

and show some basic details about the place

that was selected.

8. Adding a "Place Details" Screen (Challenge Solution 6/6)

So let's now work

on this place detail screen.

And actually the file should be named place_detail.

Not a big thing, but it is for a single place,

hence I prefer this name.

And then in there, in this file,

we'll actually change the widget tree quite a lot later.

But for the moment I'll set up a basic widget,

which I'll name, PlaceDetailScreen,

which extends StatelessWidget

because I won't need to manage any state here.

And I then expect to get a place as an input.

So using my own place model to which you

therefore also must add an import of course.

And I expect to get that place through the constructor.

So therefore I add the constructor,

accept and forward the key,
and also require and set this place.
And of course the constructor is named `PlaceDetailScreen`.
And then therefore we can now add the build method
and then return our widget tree.
And as mentioned, that will change later.
For the moment, I'll just return a Scaffold widget
with an appBar where I use the built-in AppBar widget
to set a title.
And the title here should be the title of the place
for which this screen was loaded,
which of course at the moment
is the only place data we have.
But that will change later.
So therefore, for now, I'll output a Text widget here
where I use my `place.title` like this.
And then for the body here, I won't do too much right now.
Instead I will simply use a Center widget.
And then they are also output my title again
by again using this Text widget.
So really simple, not a fancy screen right now.
That will of course change later.
I also will apply some styling here.
And for this text here, I'll use the same styling
as in my `places_list.dart` file for this fallback text.
So we can just copy the style parameter
and its value from there
and use that in the `place_detail.dart` on that Text widget.
And with that, I got the base set up.
I got this basic screen which will change later.
And now we just have to make sure
that we load this screen from inside our `places.dart` file.

So from this screen, if we tap one of our places.

And therefore it's actually the `places_list` to where we should go because there we are displaying those list items.

And now on that `ListTile`, I want to add another parameter.

The `onTap` parameter, which is a parameter supported by that widget, which allows me to register a function that is triggered when a user taps a list item.

So here I'll use an anonymous function because I will need access to the context.

And that is available here in this build method, of course, we get it here.

And then we can use `Navigator.of context` to push a new screen onto the stack of screens.

And of course we again push by using `MaterialPageRoute` here, getting our builder where we then return the widget that should be pushed as a screen.

And that widget here simply is the `PlaceDetailScreen`.

And now we must pass in the place for which we wanna load that `PlaceDetailScreen`.

And we can get that with help of the places for which we're rendering our list items.

And then by accessing a specific index, which we get here from the `itemBuilder`.

And that then is a single place that should be passed to `PlaceDetailScreen`.

So that's how we make sure that these items are tappable.

And with all that logic added, if we do tap one of these items, we're taken to this `PlaceDetailScreen`.

And that's all the functionality I want right now.

And with that, if I restart everything,

we got this basic app here where I can add places.
Like this, and where we can then also tap these places
to go to the detail screen.
And of course, with that, we're not doing anything new,
we're not using any native device functionalities,
but we practiced some of the key Dart
and Flutter concepts you must know,
and some key concepts about which you learned
in this course up to this point.
And now in the next lectures
and for the rest of this section,
we're going to dive into these new features.
And you are going to learn how you can use the device camera
or get the user's location or display a map and so on.

9. Adding a "Pick an Image" Input

So now that we got some practice again,
it's of course time to dive into
those native device features,
because ultimately, that's what this course section
here is about.
Now, in case you skipped those previous lectures,
which you of course could,
you'll find my current lib folder
attached to this lecture here
so that you can replace your lib folder
and all the folders and files in it with my lib folder
and all the files and folders in there.
And if you do that, you should also make sure
that you add some dependencies.
To be precise, `google_fonts`, `uuid`, and `flutter_riverpod`
should be added as dependencies,

either by just adding them to your pubspec.yaml file
like this.

Once you then save that file in VS Code
with the Flutter extension installed,
those packages should be installed automatically.
Or you manually visit those package pages
and follow the installation instructions you find there.

With that all out of the way, we can get started
with native device features.

And I do want to get started with taking some images,
to be precise, when a user adds a new place,
the user should be able to use the device camera
to take an image for that place.

For that, on that add_place screen here,
I wanna add a new input below my title here,
and that should be the ImageInput, let's say.

So an input that we can use to take an image.

And there is no such built in input that would allow us
to use the camera and take an image.

Instead, we'll have to build our own.

So therefore that's what I'll do,
and I'll do that here in the widgets folder.

I'll add an image_input.dart file,
which will hold the ImageInput logic.

And in there we're going to add the code to allow the user
to open the device camera,
show a preview of the image that was taken,
and then also pass that image back to the add_place screen
so that there we can use it to add it
to the place that's being created.

So inside here of image_input, I'll create a new class,
and I'll name it ImageInput.

And this will actually be a StatefulWidget,
because here we will need to update the UI
and manage some state.

Now, for the moment, I'll set the ImageInput constructor
to a constructor that only accepts and forwards the key.
So I don't accept any other parameters right now.

And I will add createState,
because of course we must add this method
to the stateful widget.

And of course we also need that state class,
the ImageInputState class,
or _ImageInputState to be precise,
where we extend State,
and that State belongs to the ImageInput, I'll zap here,
and then here in createState, we return a new instance
of ImageInputState.

And in that state class, we of course add the build method
as we always do.

Now, in that build method, I wanna return the widgets
that should make up this ImageInput widget.

And in this ImageInput widget,
I in the end wanna display a button that can be tapped
to open the camera.

And once the image has been selected,
the button should actually be replaced
with a preview of that image.

So I wanna show that image that has been picked,
though that image should also be tappable
so that we could replace it.

Therefore here what I'll render will be a container,
because this Container widget will allow me
to control the size and also the alignment

of the content inside of the container.

And here I'll set a height of 250 pixels.

I'll set a width of double infinity

to make this container as wide as possible.

And I'll add the child here.

And the child now, as mentioned,

should either be a preview of the picked image

or a button that allows us to take an image.

And therefore at the moment,

I'll just set the button as the child.

It should be a `TextButton` that also has an icon.

And then here the icon I wanna use

is of course the `Icon` widget where I use the camera icon,

which sounds fitting for this purpose of opening the camera.

Then the label is a `Text` widget

where I say ('Take Picture').

And then of course, we also need `onPressed`

where at the moment I have an empty anonymous function.

Now I wanna make sure that this button

is centered inside of that container,

which has a fixed height.

And to make sure that this is the case,

I will therefore add the alignment parameter

and set this to `Alignment.center`,

which will center the child horizontally

and vertically inside of this container.

So with that, we have this container and this button.

If we go back to `add_place`, we can now use this here.

So here I'll add my `ImageInput` widget like this.

And of course at the moment it doesn't do anything yet,

but this will change later.

If I save this and I go to this screen,

I now have this Take Picture button here which I can tap.

And as mentioned, eventually this will be replaced with the image that we did select.

Now, I also want to add a little border around this input to make it clear where it starts and ends.

And for that I can go to my ImageInput widget again and there to the container,

and set the decoration parameter to BoxDecoration, because this BoxDecoration constructive function then allows me to set a border.

And here we can then create a new border

with Flutter's Border clause,

where I could now manually configure

the different sides of that rectangle.

So the different borders on the different sides.

Or we use the special all constructor function,

which allows us to configure all the borders

for all sides in one step.

And here I'll give all my borders a width of one,

and set a color of Theme.of(context).colorScheme.primary.

And there I'll actually add some opacity here, like this.

So this now should add a nice little border as you can see.

And I also wanna add some spacing between the title input and this image input.

And therefore back in add_place, we can copy this SizedBox

and add it in front of ImageInput

and maybe add 10 pixels of spacing.

And with that, if I save this and reload,

we now got some spacing between the title input

and the image input.

So that's looking good,

but of course it's not doing anything.

Now, to do something,
I'll add a new method here in ImageInput
in the state class here.
And that will be a method which I'll call takePicture.
And in there I then wanna open the camera and take an image.
Now, we'll do this in the next lecture.
Here, as a last step,
I'll connect this function to my button
by using it as a value for onPressed.
And with that, the question now
is how we can take an image here instead of takePicture?

10. Installing the "Image Picker" Package

Now, as so often, there is a package
that can help you with a certain functionality.
In this case, if you search for Flutter image picker,
you'll find the image_picker package,
which is a package,
so a library you can use in your code
to help users pick images,
either through the camera
or by using the existing photos on a device.
Now, if you visit the site of this image_picker library,
you will learn more about it.
For example, regarding the installation,
because as it turns out,
for some libraries,
especially for all those libraries
that use certain native device features,
you often have to go through extra configuration
and setup steps to make sure that they work.
Though, we'll do this together in the next steps,

first we have to install it
by running this command here.
So back here in the project,
I'll run `flutter pub add image_picker`.
This will install this package into this project.
But now as mentioned,
in order to use it,
you have to go through some extra steps,
which are mentioned here under installation,
and definitely check out these instructions
when you are following along on your own machine
because sometimes some aspects
of this installation process can change,
and you should therefore always revisit those instructions
to make sure that this package works
as intended on your system
and in your project as well.
Now, here we, for example,
learned that for iOS,
we should add the following keys
to the Info.plist file,
which can be found in this location here,
these three keys here in the end.
So that's what I'll do.
Back in my project,
I'll search for this plist file,
which I find in the ios folder,
which you might not have
if you're following along on Windows.
You might have it,
but it might also not be there.
But if you're on Windows or Linux,

you can't build for iOS anyways,
so in that case you could,
of course, skip those instructions.
But here, I'll dive into this ios folder.
And then there as we learned into the Runner folder,
there we should find the Info.plist file.
Here it is.
And in that file, we can set up various key-value pairs,
which in the end configure this iOS app.
And here, we should make sure that we add these three keys
with some descriptions.
In the end, the descriptions we add here
will show up on the screen
if this library here becomes active.
For example, the NSCameraUsageDescription
would be the text that is shown to the user
in an automatically created pop-up
that opens up asking for permission to use the camera
right before the camera opens.
So these are in the end description texts
that will ask the user for permission for certain features,
certain device features,
that are being used by this package here.
So I'll start with the NSPhotoLibraryUsageDescription,
and I don't have that key yet,
so therefore I will add it here
inside of this dict.
We add a new key by typing key like this
between angled brackets
and then also a closing key element,
which has an angle bracket, forward slash and then key.
And then there, we paste in NSPhotoLibraryUsageDescription,

and then we have a string element here
between which we pass the text
that will be shown to the user,
Pick your favorite place image,
something like this.
Again, keep in mind this will be shown
to the user when they're asked for permission.
So when they're asked to grant your app permission
to use the image picker or the device camera
or something like this.
Speaking of the device camera,
that's the next key I want to add here.
So we add this NSCameraUsageDescription key,
and then here the text is maybe the same,
or maybe Take your favorite place image.
And then also the NSMicrophoneUsageDescription,
though theoretically, that should only be needed
if you intend to use this library for recording videos,
which is also a feature it supports.
But here nonetheless, I will add it
with that same text as before.
And with that, we adjusted the Info.plist.
So that was iOS.
Now, for Android, as you learned here,
there is no extra configuration required,
the plugin should work out of the box,
which of course is great.
And therefore, with that,
we're ready to go and we can use this library.
Now, of course, feel free to take a look
at the official README here
to learn more about what you can do with this library

and how it may be configured.

There is way more you can do with it
than we're doing with it here in this course, of course.

11. Using the Device Camera For Taking Pictures

Inside of Take picture,

I wanna use this image picker to well take an image
and for this, we should first of all add a new import
where we import from the image picker library
and there from the image_picker.dart file.

And then in take picture,

we can create a new ImagePicker object
by instantiating the image picker class, which is
provided by ImagePicker by this library.

And this will give us such an imagePicker,
which I'll store in a final variable.

And this imagePicker can then be used
to call the pickImage method.

Now as you see, there are also some alternatives
like pick video or allowing the user
to pick multiple images.

But here, it's pickImage.

And next, we have to specify a source.

Now that source must be a image source value.

So a value of type image source,
and ImageSource simply is another enum in this case here
provided by the image picker library.

This offers us different sources, two to be precise so
that we either use the gallery or the camera.

And here I wanna use the camera so
that the user must take a new image.

But of course, you could also allow the gallery here.

You could also add an extra button, one for opening the camera and one for using the gallery. But here, I'll go for the camera.

Now there are additional parameters that can be set on pick image. For example, you can control the imageQuality or the max height or width.

And here, I will indeed set a maxWidth of 600 so that the image is scaled down and we don't have super large images, which take up a lot of space on.

This can also take long to load.

And with that, I configured this image picker.

And now, pickImage will actually return a future which eventually returns an XFile.

Now what's that? Well, it's not a TV series. Instead, it is simply a file that contains the image.

An XFile simply is a type that represents that file.

So since it's a future, we could use then here to get hold to that file that is eventually yielded.

Or alternatively, we add async here and then we can await this and we'll then get our pickedImage here, which is that file.

And in the next line, we can then therefore use that image.

Now as you see, it's actually an optional value. It may be null because, of course, the user may open that camera and start the image picking process without then actually taking a photo.

Instead, the user could just close the camera without taking a picture and therefore, then,

of course, pickedImage would be null.

Therefore, here I wanna check if pickedImage is equal to null, in which case, I'll just return and not do anything else.

But if we make it past this if statement here, I know that I do have an image and in that case, I wanna show that image.

I wanna replace that text button with a preview of that image.

And for that here in this state class, we should add a new variable, which should actually be of type file, not XFile, but File because we'll need that file type later for showing a preview of that image, which I'll name takenImage or selectedImage.

Now the File type is not available like this.

Instead for it to be available, you must import dart.

And then from, there the IO sub-package.

And with that, then here we have that File type.

And since I am defining a type here, of course, as you learned, you should omit var.

Now this can also be null though, hence the question mark because initially, for example, I don't have a file, I don't have an image, but I'll set it here after we took an image.

So then I'll set selectedImage equal to pickedImage.

But now here the problem will be that pickedImage is of type XFile

and that's not compatible with that File type

I actually wanna store in selectedImage.

Hence, we must convert the XFile to a normal file.

And thankfully, that's super easy to do.

We just instantiate the file class
that is provided by Dart like this
and we have to pass `pickedImage.path` as a value to file.
So this `pickedImage` has various properties
and methods that allow us to interact with that file.
And it for example, has a path property
that is simply the path to that image file on that device
because as we take that image, it is stored on the device
and this path tells us where it's stored.
And if we pass such a path to this File class here,
we then get a File object that stores that path
and that can be used with that File type here.
And I'm doing all of that
because now, we can go down to the widget tree
and there, we can render this button conditionally
by adding an extra variable of type Widget,
which could be the content of this container.
And that could by default be our text button.
So I will cut this button and store that in content,
but if `selectedImage` is not null,
so if I have a selected image, then I want
to override content
and instead store some widget in it that preview my image.
And for that here, we can use the image widget
built into flutter and constructed
with the file constructor method
because that constructor method allows us to create a image
and show an image on the screen based on some file
of type file.
Because image wants an input which is of type file,
it then has many other properties,
many other parameters which allow us to configure

how the image will be displayed.

But it always needs such a file.

So therefore here, I'll pass a file

and the file, of course, will be my selected image,

which I know won't be null here.

So I can add the exclamation mark

because I'm checking for it to not be null here.

I'll then also add the fit parameter

to set the automatic sizing of that image

to BoxFit.cover to make sure

that it'll always look good even if it must be resized.

And it's then the content that's output here.

Now with that all done, if you save everything,

you should stop this running session here

and restart your app

because if you add a library

that uses some native device features,

it often won't work if you don't do such a complete restart.

That's why I'm doing it here.

And with that done, if you go

to this add new place screen and click take picture.

The camera should open up

and on iOS on the simulator, this will not work.

There is no camera simulator there.

You would have to test it on a real device on iOS instead.

And there, you should be open to open the camera.

And there, you should then also be asked

for permission the first time you're getting there.

And once granted, you can take images there as well,

but here on Android, you have this camera mode,

this simulator here, and you can take a picture

(taking picture loudly) and then confirm this.

And if you do so, it should show up here.

Except for that it doesn't.

And of course, it doesn't show up here

because when I am setting my `selectedImage` here

and take picture, I'm not using `setState`.

And we still need to do that even though it's an image

and we were in that camera, we still need to call `setState`

to tell flutter that the UI should be updated.

That's always important.

Now with that though, if you add this

and you go there again, you should be able to take

that picture, save that, and here it is.

Now I also wanna make sure

that it takes the full width here.

And to do that, we can go back to the image input

and there to the image file widget here.

And besides setting the fit,

I'll then also set the width to infinity.

So to `double.infinity` to make sure

that it takes as much width as it can get,

which indeed then works as intended.

And I will also do the same for the height here

and set the height to double infinity so

that it will also always take the full height of

that container, so that with that, if we save that

and I go back here

and I take another picture, (camera clicks loudly)

this looks better.

Now with that, the only missing feature is

that I wanna make this image tapable so

that it opens the camera again

and I can replace that image with a new image.

For that, I'll go back to my widget here.

And image file can simply be wrapped with another widget.

And that other widget should be the gesture detector because that is a built-in widget that allows us to listen all kinds of gestures on the child widget.

So in this case here, I can add to the on tap parameter to listen to taps on that image and then trigger takePicture like this by assigning it as a value to on tap if the image is tapped.

And now with that addition in place, if you go back here, I can take a picture of course,

(camera clicks loudly)

and if I then tap on it, I can replace it.

If I rotate the camera, which on Mac I can do by pressing the control key and rotating with the mouse, (camera clicks loudly) you see that now the updated picture is taken.

So taking those pictures works.

Now of course, they should also be stored and displayed together with our title on that list of places.

12. Adding the Picked Image to the Model & "Add Place" Form

So let's make sure those images, which we can take are also stored.

For that, I'll first of all go to the place.dart file in the models folder and there we need to add something to that model because besides storing a title and an ID there, I now also wanna store an image.

And here for me the image should be of type File using that File type which is provided by Dart io.

And I'll name the property image here.

And of course it should also be set when this place here is instantiated, then the image should be set.

Now this place gets created here in providers and there we now therefore have to get and use that image.

So here in ad place we should accept the image input.

And there I expect to get my image as a file.

Hence we also need that dart io import here in that user places provider file.

And with that we can set the image to the image we got here.

Now of course we have to make sure that in at place where we in the end call at place on our provider on this notifier clause, we forward that image that has been selected

and to forward the image, we first of all need to get it here in this ad place screen state clause because at the moment we only have the image input in there and we have that image inside of the image input in that selected image property here.

But we're not getting that image from this image input widget to the at place screen widget.

And we of course need to do that.

Now to forward the image from this child widget to the parent widget, we can use an approach which we have used many times before in this course in the child widget in image input, we accept an extra parameter which will store in a property and the property will be of type function because I want to get a function here as a value in this constructive function.

Now the function which I expect to get, should be a function

that returns nothing

but which takes one argument, one parameter of type file,
the image that we should forward.

I'll name this property here on select image

or on

pick image.

And I expect to get that function,

which should be stored in this property

as a value here in this constructor function for one

of my parameters, for a required parameter to be precise,

a required parameter of the same name.

Now it's this on pick function which I want to execute here

and take picture

after we took an image to let the parent widget,

which will pass the function

for this on pick image parameter

and property to image input.

So therefore it's that function defined in the parent widget

which we'll receive here and which I want to execute here

and take picture once we took an image.

So therefore here at the end we can call on pick image

or to be precise widget on pick image since this is defined

in the widget class that belongs to this state here.

And we get access to that class

through the widget property as you learned.

And we can forward our selected image here as a value

to on pick image.

Now selected image theoretically could be null though we

know that here it won't be null

because we're setting it equal to this file,

which is based on the picked image, which will not be null.

Hence here we can add an exclamation mark.

With that, in add placed dot dart,
we can add a new variable
to at place screen state
the selected image, which will be of type file.
Hence I'll set that type here and it may be null.
And for file to be available, we of course as always
must import from DART io.
And with that import added, I'm defining this property here.
And now we just need to update this
whenever an image is picked.
And for this here on image input,
we must set this newly added on pick image parameter
and set it equal to a function.
A function that will get
that image from image input in the end
because that is in the end
where we will be executing this function.
We're executing it here and we're passing the image to it.
And therefore here I get this image file.
And in this anonymous function,
I therefore just have to set selected image equal to
that image which I am getting here.
We don't need set state here
because in here I'm not doing anything
that should be updated.
The UI doesn't need to be rebuilt just
because I picked an image,
because the preview is built here in image input.
And here I am using Set State.
But in the add place screen there is no need
to rebuild everything because I'm not updating anything just
because an image was picked.

Instead, this is all that needs to be done.

And with that we have the selected image available here
and therefore now here in save place we can use
that selected image to send it to add place.

Of course, we also might want to add some validation
and check if enter title is empty
or if selected image is equal to null,
in which case we wanna return
and not continue with this step.

Otherwise we do want to continue.

And in that case, we should also add an exclamation mark
here because we know with certainty
that selected image will not be null and
therefore we should also let Dart know about that.
And this will now pass the image file to add place.

13. Previewing the Picked Image

Now, with our image added here
to our Place model
and in the provider,
we can of course also display it on the screen.

For example, in places.list.

There I have this list tile
where I output my title.

And there I now also wanna display the image.

For that, I'll set the leading parameter on ListTile,
which is one of the built-in parameters that can be set.

And this parameter wants a widget as a value,
which will simply be shown at the start of the row.

Now, the widget I do want to display here
is the CircleAvatar widget,
which simply renders a circle

with some content inside of it.

And here we can set the radius of this circle,
and I'll set it to 26.

Though you can of course play around with that.

And I'll set the `backgroundImage` parameter here
to basically tell Flutter
that inside of this circle,

I wanna have a image in the background.

And I could then also add some foreground content.

But in here, I only wanna have an image,

so in the end, it also doesn't matter

whether I set foreground or `backgroundImage`.

This would only matter if the foreground content
would be transparent, for example.

But here set my `backgroundImage`.

And this now wants an `ImageProvider`,

so not a widget,

hence we'll not use `image.file`

because that would give us a image widget

and `backgroundImage`,

instead wants an `ImageProvider`.

Instead we'll use `FileImage`,

which sounds similar,

but which is such a `ImageProvider`.

It extends `ImageProvider` in the end, as you can see here.

And it's this file image, which I'll set here,

and the file which I wanna display,

which is this first argument

wanted by a file image,

is simply that file stored in my place.

So here we can then use `places` index to get a single place,

and then there the image,

which is such a file.

With that, we should be able to add a new place,

give it a title, take a picture,

and click Add Place.

And we see that image here as well.

Now, we might want to add some padding around that list

to add more spacing between this top edge of the image

and the app bar.

And for that I'll go to my places.dart file

and wrap my PlacesList here

with a Padding widget

to add some padding in all directions, maybe eight.

So now we have that extra spacing.

More importantly,

I wanna make sure that when we tap this,

we're taken to this detail screen

where I now wanna display this image.

And indeed here my idea

is to display this image

as a pretty much full-screen image.

And then later, on top of that full-screen image,

I want to display a smaller image

that shows us the location on a map

and the address as human readable text.

But at the moment, I just wanna have that full-screen image.

But since I plan on displaying some content

on top of it later in place_detail.dart,

I will replace this center widget,

which displays the title,

with another widget,

because the title is already shown here in the app bar,

so there's no reason to also show it on the screen.

Hence I'll remove it,
and replace it with the Stack widget instead.
About which you learned earlier in the course already.
The Stack widget is simply a widget
that allows you to display multiple other widgets
so that they are really placed directly above each other,
on top of each other,
that they overlap each other.
And here on this stack,
we have to set the children parameter.
And now add all the widgets
that should be placed on top of each other,
starting with the bottommost widget.
And that should be this image.
So here I'll use Image.file again,
because here I wanna display a Image widget.
And the file I wanna use here
is of course my place.
So place.image.
And I'll then configure this image
and set the fit to BoxFit.cover,
and set the width to double infinity
to take as much space as possible,
and the height to double infinity
to take as much space as possible.
If I do that, I do see this image here full screen
below the app bar.
And as mentioned later,
we'll put some content on top of that.
But for the moment, that's it,
and we're now ready to move on to pick a location.

14. Adding the "location" Package & Starting with the "Get Location" Input Widget

So now that we can take a picture here,
if we want to,
let's make sure that we now also add a feature
that allow users to add a location
to the place they're creating.
Now, to allow users to pick a location
or to be precise
to automatically get the location of a user,
we'll use yet another Flutter library,
another package which we can use.
And that would be the location package.
This is another package that exists
to make using certain native device features,
like getting a user's location,
a bit easier
so that you don't have to write any native source code
to make that work in your Flutter app.
And as with all those packages
that unlock certain native device features,
you should therefore always, as mentioned before,
take a look at the official installation
and setup instructions.
Now, of course we'll start by installing the package
with this command here.
So that's what I'll do here
in my project, running `flutter pub add location`.
But once this was installed,
you should, as mentioned,
take a look at the README
and go through the extra setup steps
that might be required.

Now, here you, for example, learn that for Android, generally, everything should be set up automatically, though if you would be building an app that uses the location in the background. So whilst the application is running in the background, you would need to make some extra adjustments. Now, for iOS, we again have to add something to Info.plist. And now here it's basically this key which we should add. This one here is deprecated, and this one here should only be used if your app really needs to access the location at all times, even if it's not running. So for our app here, it's only that key that must be added. Hence back in this Info.plist, we should add this key here, and then add our text that will be shown on the screen. And here I'll say, "The location for your favorite place." Anything like this. With that, that should be all for Android and iOS. And we're now ready to use this package. Now, as always, of course, definitely take a look at their usage instructions on the official package page to learn all about this package and all about the different things you can do with it. And here we, for example, learn how we can get the location of the user.

And you, for example,
learn that you should always go
through some extra checks
to see whether this location service
is enabled on the device.
So if you can technically get the user's location.
And you should also check whether you have permission
to get the user's location.
So therefore I'll get back to this code later,
because for the moment,
I will first of all add a new input widget,
a `location_input` widget, and a `location_input.dart` file,
because I wanna add this as a new input element
to the `add_place` screen,
just as we added the image input there.
And therefore here in `location_input.dart`,
I'll build this new input widget.
For that, in there, we can add `LocationInput` as a class.
And I wanna extend the `Stateful` widget here
because we will need to manage some internal state.
And then, of course, you know how that works.
We add a constructor function,
where at the moment I only forward that key.
I add `createState`, as we always do,
pointing at location input as a state.
And we add that state class,
the `LocationInputState` class,
which extends state connected to the `LocationInput` widget.
And then here we return a new instance
of our `LocationInputState` class.
And then down there, we add the `build` method.
And in that `build` method,

we now output the widgets
that should make up this location input.
Now, as always, it is of course up to
you how you wanna build this,
but here I'll build it such
that I'll return a column widget
because I wanna have multiple widgets
above each other along the vertical axis on the screen.
And here I essentially wanna have one container
that will be used for showing
a preview of the place that was picked.
So basically a map snapshot, you could say,
where I can see that picked place.
And I'll get back to that container soon.
And below that container,
I wanna have a row
where I'll have two buttons next to each other.
Two buttons because one button
should use this location package
to get the user's location automatically.
The other button should later open a map
where a user may pick a place manually.
So therefore here in this row,
I'll add two text buttons,
which will actually also display some icons.
So I'll use that icon constructor function.
And then for the first button,
I'll set the icon to a new icon
using the icon widget of course,
where I wanna use the location_on icon.
And the label should then be a text widget,
a const text widget,

where I say, "Get current location," for example.

And of course we also must add `onPressed`

and set this to an empty function right now.

And below that, I have another text button with an icon,

where the icon should actually be the map icon,

which looks like this.

And the text could be, "Select on map."

Now we got these two buttons next to each other,

and to row, I'll also add the `mainAxisAlignment` parameter

to control how these buttons should be positioned

along the main axis.

And here I'll choose `MainAxisAlignment.spaceEvenly`,

which simply adds an even amount of space

around these items in the list.

Now, with that, if we go back to `add_place.dart`,

we can add this maybe below the image input

and maybe with another-sized box in between.

And then there we can add the location input,

like this.

And if we save that and reload,

we should be able to go there.

And we see these two buttons.

Now, we don't see the container

above these buttons yet,

so that's something I'll change.

Back in `location_input.dart`,

I wanna work on this preview container.

And there I'll give this a fixed height of 170 pixels

and a width of double infinity,

and then the same decoration

as I had on my image input.

Hence we can copy it from there

because I just wanna make sure
that I have the same border.
I'll just copy it here from the image input
and add it to this container here.
And then I also wanted to find the child widget,
which should either be that map snapshot Basically,
once we did select a location.
and until we have that,
I simply wanna output some text
where I say, "No location chosen."
And at the moment, that's the only thing I'll output here.
And I'll add `textAlign` here
and set this to `TextAlign.center`
to make sure that the text
will be centered horizontally.
And I'll set the style here
and set this to Theme of context `textTheme.bodyLarge`
with an exclamation mark
to then copy it with some extra settings.
And the extra setting I wanna add here is,
as so often, the color,
where I'll again reach out to the theme
and then the color scheme there.
And on that color scheme,
I'll choose `onBackground`.
Now, with that, this looks better.
I also want to center this text vertically.
And to achieve this, we can go to this container here,
and of course set the alignment
to `Alignment.center`.
With that, the text is centered.
And now, of course, the next step is to make sure

that when we press this Get Current Location button,
we actually do fetch the user's current location.

15. Getting the User's Current Location

Now, to fetch the user's current location
when this button here is pressed,
we can add a new method here to `LocationInputState`
and I'll name it `getCurrentLocation`.

And in this method,

I now wanna execute the code you can see here
on this location package README page.

I'll just copy that code
and paste it in there.

And since this code uses `await`,

I'll add `async` here.

And to unlock these special types, like `location`,
which are provided by the location package,

I'll, of course, also add an import,

where I import this location package,
and from there, the `location.dart` file

so that all these extra classes and so on are available.

And this new keyword here is actually not needed anymore,
that is some old Dart code.

Now, here, I'll also rename these variables

to not have leading underscores

also in all the places where they are being used,

just to be in line with all the suggestions

I'm getting here in Visual Studio Code.

And that should therefore then give us this location data

in the end here,

after checking whether the service is available

and we have appropriate permissions.

Now, this location data which we're getting in the end will be an object that will have a couple of properties. For example, it will have a latitude and longitude property, which is simply, guess what? The latitude and longitude of that picked location. And it's this location information which I then also wanna use to generate such a map snapshot and also to get the human readable address of that location because latitude and longitude alone might not be too helpful, and that's therefore what I wanna do as a next step. But before we do that, I at least wanna show if it technically would work to get a location. So for that here, I'll print my location data and then there the latitude and also the longitude to see if I am getting some values here eventually. And I also wanna show some loading spinner whilst we are fetching the user's location. So for that here, in this `_LocationInputState` class, we can add a new variable, the `pickedLocation`, and I'll set this to type `Location`, which is this type provided by the location package, the same type this location data is of. And it may be null initially, hence this question mark. But I'll also add an extra variable, `_isGettingLocation`, which initially is false, but which I'll set to true once we start getting the location. So here instead of `_getCurrentLocation`, there I'll set `_isGettingLocation` to true. And of course, this should be wrapped with `setState` here

to make sure that the UI can be updated
and the build method is re-executed.
And I'll actually not do that here
now that I think about it,
but instead after these checks
so that I really only show that spinner
once we are starting to get the user's location.
So right before calling getLocation here,
which can take a while,
getting the user's location can take a while.
So then here I'll set `_isGettingLocation` to true
and I'll set it to false once we're done,
so right thereafter.
But again, as mentioned, this can take a while.
And now we can use `_isGettingLocation`
to show a loading spinner,
and we might wanna show this loading spinner
instead of these buttons
or maybe inside of this preview container.
And that is indeed what I'll do.
I'll go to my build method here
and add a new variable `previewContent`,
which by default should be this text here.
But then I'll add a if check
and check if (`_isGettingLocation`) is true,
in which case `previewContent` here
should be such a `CircularProgressIndicator`
that can be a const widget.
And it's then this `previewContent`
that should be shown inside of this container.
And now with that, we just have to make sure
that `_getCurrentLocation`, this method here,

is called when we click this button
so we should set it as a value for onPressed here.
And if we do that
and save that and reload everything,
if we go here and I click Get Current Location,
I again get an error if I take a look at my debug console.
And the reason for that
is that as mentioned before,
I do recommend that you stop the running debugging session
and you restart your app
if you add a package
that uses native device features,
like this location package does.
With that, once it restarted
and you then try again,
you should now be able to see this permission pop up,
which, of course, might look different
depending on the device you're on,
where you now have to grant this app permission
to pick a location.
So here, I will grant that permission,
and you see that loading spinner.
And then, of course, we still see no preview here
because we haven't added that functionality yet.
But if we take a look at the debug console,
there, we should see
some latitude and longitude coordinates.
So that's working.
And as you saw, asking for permission worked
and we did check whether we had permission or not.
But with all of that, we were able to pick a location.
And of course, this also works on iOS.

If you run the app on a real device there,
you can now not just take a picture,
but also pick a location.

16. Using the Google Maps API - Setup

Now that we're getting the user's location,
the next step is to translate that location,
which is a pair of latitude and longitude coordinates
along with some other data into a human readable address.

So an address that makes sense to humans.

And in addition, I also want to generate a map snapshot
that we can display here as a preview
after a location was chosen

here for this preview content widget.

Now for both aspects,
we're going to use Google Maps here
simply because it's probably the most common used solution
for getting addresses or showing maps.

If you wanna use the Google Maps API,
you will need a Google account
and also a credit card, unfortunately.

Getting started with these Google Maps APIs
is possible for free,
so you won't get charged
because you have a convenient free tier,
some credits which are granted to you by Google,
but you will need that credit card.

Alternatively, if you don't have a credit card,
you will unfortunately have to follow along here
without adding those maps features.

But with all that, if you go to the maps platform page
by searching for Google Maps API,

you can click that get started button
and then log in with the Google account you want to use.
And once you're logged in
and you are viewing the Google Maps platform page here,
which might look slightly different
but should look something like this,
you should theoretically be asked to
connect a billing account
and you might need to create such account first
before you can get started.
And thereafter, you should be presented with
such an API key.
You also get that \$200 budget for free by Google
so that you can effectively use
these Google Maps features for free,
at least to the extent
we're using them here for this course.
You can then also choose which Google Maps API
should be unlocked.
And here I'll choose all
and then go to this Google Maps platform.
And here I'll choose maybe later.
And if you did not get this prompt after going there,
you can also always manually go
to the APIs page of the Google Maps platform page here
and there you can manually select APIs
by clicking on them and enable or disable them.
Though of course, if they aren't enabled,
you will not find them here under enabled APIs,
but down there under additional APIs.
And for this course here, for this section,
we will use the places API, the Maps Static API,

and also the Maps SDKs for Android and iOS.

So those APIs should all be enabled

and will need that API key that was shown to us.

And if you did not save that, that's no problem.

You can go to credentials here

and there You will also see that Maps API Key

and you can show it again.

And here you can also create a new API key

by clicking on Create Credentials and then API key.

And we'll need that API key

to use these Google Maps features

from inside our Flutter app.

And now with all of that out of the way,

we're going to get started

with some of these Google Maps features.

17. Using Google's Geocoding API

And now we will get started

with the Google Maps and the features

provided by this Google Maps collection of features

by using the Geocoding API, which is one

of those features offered by Google,

under that Google Maps collection.

The Geocoding API can be used to translate latitude

and longitude coordinates into a human readable

address you could say.

So, an address that means something to users.

And for that we can use the reverse geocoding approach

because that allows us to do exactly that,

translate that into a human readable address.

And for that you can of course also explore the official

documentation to learn which kind of request you need

to send to Google,
because that is what you will need to do in the end.
You will need to send an HTDP request.
But that's of course also what we're going to do together.
And for that, to create that location here,
I will go back to the location input .dart file
and hereafter picking that location,
I also want to send this request
and translate this location into such a
human readable address.
And for this we'll need to send an HTTP request.
Now for that, in turn, we'll need to install
that HTTP package.
So we should run `flutter pub add HTTP` to use
that HTTP package about which you already learned in the
previous course sections.
With that installed,
we can import it here into this location input
dart file `package:http`
and then there `http.dart` as `http`
as we did it before in this course.
And with that imported down here,
maybe before setting `isGettingLocation` to `false`,
we can create a new URL by using `URI`
and then either `HTTPS` again as we did it before,
or the `parse` method,
which takes a full URL, a full URL string
to be precise and then converts it into such a `URI` object.
And that's the approach I will use here.
And the URL to which we should send a request
can be found here in the official documentation.
If you take a look at the example down here, it's this URL,

which we want to use.

But here we want plug in our latitude and longitude
and also our API KEY, which will be used for authenticating
that request and telling Google
that we may make this request.

So therefore, here as a first step,

I will actually extract that data.

I will get my latitude

by accessing `locationData.latitude`

and then the longitude by accessing `locationData.longitude`.

And then we can replace this part here,

40. something in front of the comma

but without replacing the comma

with dollar sign and then lat

and then the minus 73 part here,

in my case with dollar sign lng

and the comma between the two must stay there

don't delete that accidentally.

And then for API key, you can plug in your API key.

So in my case, this key here, which

by the way won't work anymore when you are viewing this.

So, you don't need to try using that.

So with that we have the URL

to which the request should be sent.

And with that URL setup here,

we can next send the request by using the HTTP package

and then the GET method,

because it's a GET request that should be sent here

to this URL.

And if we await this,

we will in the end get this response object.

Now the shape of this response object can be seen in the

official documentation here in this example section.

There you see that what you get back will be this JSON data

where we have a results key, which contains a list

with a nested object, so to say,

where we then have address_components,

which contains another list

where we then have different items

that identify the address.

Now you can also see if you scroll down

that in this object here, which is inside of

that results list,

we don't just have these address_components,

but if we scroll down,

we also have this formatted_address field

which contains the complete address.

And that is what I'm looking for here.

Therefore, to retrieve that, we have

to decode this response, which we can do

with the JSON object provided by dart:convert.

And on JSON, as we did it before, we can call decode

and pass our response.body to it, like this.

So this will give us our data, our response data,

and then on this response data object,

which has this shape here.

So in dart it will be a map of this shape here

with a nested list and another listed map inside of it.

We can access our address

by diving into resData.

Then there the results key,

because that is the top most key we have in this map,

which then gives us a list.

And there in this list we want the first list element,

since this will be this nested map here.

So we use index zero to access this first element.

And then, `Deer`, inside of this nested map, as mentioned, it is this `formatted_address` key which we want to access.

So that is what we use here,

and this should give us the human readable address.

And now of course, the goal is to save this address together with these coordinates in some object, which we can then later use, for example, on this place detail screen.

And that's therefore what we'll do next.

18. Storing the Location Data in the Model

So now that we got

the human readable address,

I also want to be able to save it.

And for this back here in models

where we have our place model,

I actually want to add a second model

and I will name it `PlaceLocation`.

And this should be a blueprint for a location

because in my place, just as I'm saving a title or an image, I also want to save a location.

And now a location should be a combination of a couple of properties.

To be precise, it should contain a latitude,

which should be of type `double`,

so a number with decimal places.

And I therefore also want to have a longitude.

And in addition, I want to have that human readable address,

so this string which contains

the address we fetched from Google.

Now all these properties should be set
with help of the constructor.
So here we have the PlaceLocation constructor.
And this constructor then can take a couple
of required named arguments where we set the latitude
and also the longitude,
and also the address like this.
And now in my place down there,
I also already want to add
a new final PlaceLocation type property,
which I'll name a location,
which also should be set here through the constructor
so that we can store this location as part of our place.
Now with that backend location input,
I now want to create such a custom place location,
and I want to use that place location
instead of this location type,
which is coming from the location package
for my picked location here.
So here I'll use PlaceLocation instead
using this custom class,
which we just defined in the models folder
in the place.dart file,
which therefore also must be imported.
And now the picked location is this kind of location.
I also want to update picked location down here
after getting the location and the address,
so that I do indeed set my picked location
equal to a new PlaceLocation object
where I set the latitude to lat, the longitude to lng,
and the address to the address we fetched here.
Now here I got an error because latitude could be null

and the same for longitude, and therefore indeed here,
I will add an extra check
and if lat should equal null or if lng should equal null,
I will return here.
We might also want to show an error here,
just as we also might want to handle error scenarios
for this HTTP request here,
but to keep this app a bit simpler
and focus on the core topics, which is the usage
of native device features, I will not do that here.
Instead, I'll add this basic check here
and then create my PlaceLocation down there
and store that in a pickedLocation property.
And with that,
we now don't just have the latitude and longitude,
but also this human readable address,
but that's still not enough to show a preview.

19. Displaying a Location Preview Map Snapshot via Google

Now, to show a preview,
we'll use the Google Maps Static API,
which is simply another API offered by Google
under that Google Maps umbrella, so to say.
And that's a very simple API,
which we can use to create snapshots of locations on a map,
so images of a certain place on a map, you could say.
Now, on this page here,
you'll find an example for the kind of URL
you must construct to generate such a snapshot
because it's basically an image
that is created on the fly
based on the configuration you set up in this URL here.

And I'll copy this URL therefore,
and I'll use that for generating such a preview.
And we don't need to generate the preview here
where we get the location.
Instead, as I just said, it's basically a URL
that directly leads to an image
that will be created on the fly.
Therefore, we can directly output that image with a widget.
And I'll simply add a getter
which will contain this URL here,
so that we don't need
to put this long URL into our widget tree.
Instead, here in the `_LocationInputState` class,
I'll add a new getter that will return a string
because in the end it simply will return this URL
with all the necessary data plucked into it.
And I'll name my getter `LocationImage`
because in the end, again, it will provide a URL,
a link to an image.
And I'll simply return that string here
as a string, of course,
but now we have to plug in all the necessary data.
And of course that's a very long string
and you should make sure that it's all in one line,
but we will also shorten it here, no worries.
Because for example here,
this center part,
which here for me by default is sent
to `Brooklyn+Bridge,New+York,NY`,
should be replaced
with the latitude and longitude coordinates.
So for that here in the getter,

we can get the latitude
by reaching out to the picked location
and accessing the latitude there.
And the same, of course, for the longitude,
which we also get from the picked location
and then longitude.
Now, here, I actually can be certain
that this will not be null
because I'll only use this URL if we are showing a preview,
which we will only do down there,
at least soon, if we have a picked place.
But we could also add an extra check here
where we check if (`_pickedLocation == null`),
in which case we could return an empty string here
if we wanted to,
then we can be super certain
that this will not be null.
So with that, we got latitude and longitude,
and this should now be injected into this string
after center here,
so that we have lat, comma, and then longitude.
Now, the zoom level, I'll set this to 16.
And for the size here, I'll keep 600x300.
This simply defines the size of the image
that will be generated,
and it's width times height.
Now, for the map type,
we wanna keep a roadmap,
but now for the markers,
we can delete a lot of stuff here.
With that marker's configuration here,
you simply configure which markers

will be part of that image,
so you control which pins will be on the map.
And here, I want to have a marker,
but only one.
Therefore, you can delete all other `&markers` parts
with all their configuration except for the first one,
and that will make that URL way shorter.
So everything up to `&key` is deleted here
because we will still need to add our API key here later.
And then here, for that one marker that we kept,
you can choose the color
and you can choose from those base colors
like red, blue, and so on.
So here, I'll pick red.
Now, the label here simply defines which character
is shown in the marker,
and I'll simply pick A.
And then thereafter you have this `%7C` separator,
which you should keep.
But then you again have the latitude
where the marker should be placed,
and that, of course, should be the same latitude
as is the center of this place,
and the longitude.
And here I just saw that after center,
we should have an equal sign.
But with that back on the marker,
that should be the marker configuration.
And now we just also need to add the API key,
which we also are using down here,
so it's the same API key.
Copy that and inject that here.

And then we can delete this signature part here,
it's not required.

And with that, we have a getter
that generates such a snapshot URL
that should be usable as an image.

And therefore as a next step,
we can go down to the widget.

And there I now wanna check
if we have a `_pickedLocation`,
so if this is not null,
and in that case, `previewContent`
should be set equal to an Image widget
using the network constructor
because we're now fetching an image over the network
via that URL that is returned by our getter.

So here I'll refer to my `locationImage` getter
that we just added,
and I'll add `fit : BoxFit.cover`,
and make sure that we use the full width
with double infinity
and height with double infinity
inside of that container that we're using.

And the container we're using
is, of course, that container
that has `previewContent` as a child.

And with that, we should be showing a preview of our place
after it was picked.

Now, at the moment, we won't really be able to test this
because we have a couple of errors in our code
because we updated that place model.

And to be able to test this,
I will actually revert those changes, at least temporarily,

and comment out the parameter
and this property in the Place class
so that the rest of the app will work,
and we can check whether fetching
and previewing a location here works.
So back here in the app,
if I click Get Current Location,
it starts to fetch.
Let's see if it eventually finishes.
Looking good,
it shows a location.
And by the way, on the Android simulator,
the default location always
is basically the main Google building in Silicon Valley,
so that's what we're seeing here.
But that all seems to work.
And if I open the debug console,
I also see no errors related to that.
So this all is looking good,
and we therefore are able to fetch and preview a location.
Now, we can make sure
that we're also using this location in the rest of the app.

20. Using the Picked Location in the Form

Now to use this location
in the rest of the app,
we can go back to place.dart and comment this parameter
and this property back in.
And then with that go to our user_places provider
where we create such a place.
And here we now of course need to get that location.
The location type should be PlaceLocation.

So our custom type based on our custom class
and I'll name it location here.

And with that we can then set this location
to that location we're getting here as a parameter.

But with that added, we of course have to make sure
that in `add_place` we get that location
and we then pass that to `add_place` from our provider.

And for that I want to use the same approach
as for the image,
I'll add a new property here,
which is of type, `PlaceLocation`.

And I add a question mark because it may be null
and I'll name it `_selectedLocation`.

Now this should be set
whenever a new location is selected in the location input.

Now of course, as before for the image input
to make the parent widget aware of the fact
that a new location has been picked,
I'll add a new property here to my location input widget.

And this will be of type function
and it will be a function
that receives a location
which will be of type `PlaceLocation`.

And I'll name it `onSelectLocation`.

This is my property name of course,
an equally named parameter should be accepted
and set as a value to this property like this.

And now inside of the state class,
we can use this widget property
and the function that's stored in it
after getting our location
to call widget `onSelectLocation`

and forward that `_pickedLocation` which we're setting here.

And here we know that it won't be null

because we are creating it right here,

so it won't be null here.

With that we're forwarding that location

and an `add_place.dart`.

We now should add a new function to here on `LocationInput`

the `onSelectLocation` function in the end.

And here we'll get our location of course automatically

provided by the `LocationInput` widget in the end.

And it should also be stored like selected image

in a variable in this class here.

So in this `_selectedLocation` variable here to be precise.

And therefore here in `onSelectLocation`

we can and should set `"_selectedLocation = location;"`.

And now it's this `_selectedLocation`

that will be used here when we call `addPlace`

on the provider notifier,

which we'll use as a value for that location here and here.

I also add the exclamation mark

to make it very clear that here this won't be null

and of course to be certain that it won't be null.

I'll add some validation here.

And return if selected location should be equal to null.

21. Outputting the Location Data

With that, we're adding this place

with the location.

Now we can also output it.

And I will start here in places list.

There in this list tile,

I don't wanna show the map,

but I do want to show the human readable address
and I wanna show it as a subtitle below this title.

Now conveniently ListTile

has a subtitle parameter that can be set
and it can be set to a text widget, for example.

And here I then therefore output places index.

And then there I access the location

and then the address,

which is this human readable address.

Now I also wanna have some styling here,

and for that, we can copy the style from the title text

and use that here in this text widget,

which we use for the subtitle.

And here, I don't wanna use titleMedium,

but instead bodySmall, this space text theme.

But then also apply the on background color to it.

With that, if you save this,

we should be able to add a new place, Google Office,

where we of course, take a picture

and then choose a location

at the moment by getting the current location.

And if we click add place, that's added

and we can see this human readable address here.

So that's looking good.

Now on the detail page,

I now do want to show this map snapshot again,

this image preview of the location.

And for showing that on the place detail screen,

we should go to the place_detail.dart file.

And there in this stack, I'll add another widget,

the positioned widget,

which we also saw before in the course,

which is used for positioning widgets inside of that stack
so that you control where it should be positioned
on top of that other widget.

And here the child I want to position
will actually be a column widget
and we can and should add some positioning parameter values.

And here all set bottom to zero
to make sure that this column of widgets,
which I'm about to add,
starts at the very bottom of this image.

So of this stack.

But since the image currently decides
the width and height of that stack,
it will be at the bottom of the image in the end.

And I wanted to span the entire width from left to right,
which can be achieved by setting left and right to zero.

So that in the end, this column,
which we're about to add here,
touches the bottom at the very bottom
and the left and right on the very left and right edges.

Now in this column, we of course need children.

And here, the children that should be added
should be one circle avatar
that should display this map preview.

And then below that, I want to have a container
and I'm using a container
for styling and positioning purposes,
where in the end, I wanna output some text.

And that text should be that human readable address again.

And for that here, if we start with the text,
the text should be place using this place property,
which we have here,

.location.address,
and some styling, which I will again copy
from places_list.dart.
I'll copy that style here.
And back on place_detail.dart,
I want to use that style for this text here,
for this address text.
But I don't wanna use bodySmall here,
but instead titleLarge.
Now on the container that contains this text,
I want to add alignment and set this to alignment center,
to center it horizontally and vertically.
And the text here should also always be aligned
in the center by using TextAlign
and then TextAlign.center.
And back on the container,
I want to add some padding
with const EdgeInsets.
And then here I'll actually not use all,
but instead symmetric to set a different horizontal
and vertical padding.
Horizontal padding here for me should be 24,
vertical padding could be 16.
Last but not least, I want to add some decoration here
by creating a box decoration object.
And this can be a const object
because I want to add a gradient,
a little background gradient behind that text,
this address text,
which we're in the end, outputting here.
And for that, I'll create a linear gradient here
as we also did it before in this course.

And for the colors,
I'll go for colors transparent as a starting color
and end at colors black 54,
which is a slightly transparent black.
You could of course, also pick your own colors.
But with that, I'll simply add
that there is a slight transparent
black background behind this text,
which will also ensure that this text stays readable.
Now on this gradient,
then also set begin and end
and begin should be alignment top center
so that we become more transparent towards the top
and end should be alignment bottom center.
And with that, we're outputting and styling the address.
Back in the circle avatar, I wanna output that map.
Now for that, we'll need that same trivial link
that we also used in location input.
So I'll simply go there
and grab this gather.
We could also outsource it into a separate file,
but here I'll simply copy and paste it
and paste it here into place detail as a gather.
And I'll keep the name, location, image,
but of course, we don't have a picked location here,
we just have a place
and it will never be null here per definition
because we don't allow null input values.
So we don't need this check.
And instead I can just extract my latitude
and we don't need the exclamation mark
and also the longitude,

but of course, not directly from the place,
but there from the location property instead.
So we should access the location property.
With that, we get the location image though.
And now we can use this location image
here in the circle avatar.
There I'll set a radius of 70 to create a perfect circle.
And then we can again set background image here to an image
this time using the network image provider,
which takes a URL as an input.
And with that, we should be displaying
this preview snapshot.
Now, with that, if you save this,
you should already see it as this updates here
that you have the address at the bottom
with this slight transparent black background,
which is fading out towards the top.
And you have this circle avatar that shows you the location.
And that's the look I want to have here.
Now soon this should also be tapable to open a map
where we can view this place.
But before we add this functionality,
I first of all wanna make sure
that this select on map button also opens a map
and allows us to pick a place on that map.

22. Installing & Configuring the Google Maps Package

Now to display a map, a Google Maps based map,
to be precise, we'll use yet another package,
the Flutter Google Maps package,
which has its own package page like all these packages.
And as always, of course we need to install it,

but you should always check these usage instructions,
so that you make sure that everything is set up correctly.

For example, here, it also tells you

that you need such a Google Maps project in the end,
such an account, and that the maps SDK for Android,
and the maps SDK for iOS should be enabled.

You also need such an API key,

but of course we got all these things already
because we already worked with Google Maps
before in the previous lectures.

We'll also have to go

through some additional setup steps though.

But we'll do that after actually installing the package,
which I'll do as a first step by copying this command here.

And then in our project here,

we can run Flutter Pub at Google Maps Flutter.

And with that installed,

we can now go through the remaining setup steps.

Here, we can start with Android, and as you see here,
we must adjust the minSdkVersion in the Android folder.

And there the app folder,

and then the Build Gradle file to set it to 20.

So if we search for the Android folder in our project,

and then they are in the app folder,

and then this Build Gradle file here, we can scroll down
until we find this Android part here.

And then in there the default config,

where you should set minSdkVersion to 20.

That's the first step.

Now, as a next step, you must specify

your Google Maps API key

in the Android app source main AndroidManifest.xml file.

For that, you can copy this metadata thing here
from this example.

And then go to Android app source main, AndroidManifest XML,
and there directly below application.

As you can also see here, directly under application,
you add this metadata key.

And here you must add your API key,
which in my case I'll simply grab
from this place detailed Dart file.

Because there I have it in this Gator URL here.

Of course, you must use your key, and save that here.

Don't replace API key here, just here in the value part.

And with that, we should be done with the Android setup.

Now for iOS, we also must adjust some files
here in the iOS runner folder, the AppDelegate.m file.

Or if you don't have that in the iOS runner folder,
the AppDelegate.swift file with this code here.

So here in my case, in the iOS folder, in the runner folder,
I have this AppDelegate.swift file.

And now the content in this file here
should be replaced with this content.

So that's what I did.

And here we also must plug in our API key.

So in my case, I'll grab it from the Android manifest again,
and now place it here.

With that, we should then be good to go
to use the Google Map widget in our widget tree
as you see here.

23. Adding a "Map" Screen

So now with the Google Maps package set up,
I'll add a new screen actually

because I wanna open this map as a full screen map.

And hence here in the screens folder, in the lib folder,
I'll add a map.dart file.

And in there, I'll add a new MapScreen class,
which extends StatefulWidget actually
because I want to be able to pick a place there as well.

And at the same time, I also want to be able
to show an already picked location.

And therefore here, I'll expect
to get a PlaceLocation value.

So I will create a property location,
which is of type PlaceLocation,
and of course you must add that import
to the place.dart file in the models folder therefore.

And this should be set through the constructor.

So here for the MapScreen,
I, of course, get and forward that key as always.

But in addition, I also want to get such a location.

However, here I actually won't require this location
because whilst I of course will need a location
to show an existing location,
if we open this MapScreen from the place detail screen,
I don't need a location if we're picking a location
for the first time from the add place screen
so if we clicked on this select on map button here.

So therefore, I'll actually add the location parameter,
but I'll not add the required keyword,
but instead set this equal to an initial location
by adding an equal sign here to assign a default value
to this parameter,
which will be used if no other value is passed in instead,
which would take precedence though.

And here, I'll then create a new place location object
where I'll set the default location to 37.422
and then the longitude to minus 122.084
which should be the default location
of the Google offices.

And for the address, I'll add an empty string
because I don't really care about
that human readable address here.

So that's now this default location which is set here
and which will be used if no more specific location
is passed in for this parameter.

And this simply allows us
to use the MapScreen in different scenarios.

One scenario where we have a location
and one scenario where we don't have it yet.

Now of course, besides adding all these things,
we also need that good old create state method
where we create the state for this map screen.

And then also that second class,
the _MapScreenState class, which extends state
that belongs to this widget here.

And we return a new instance
of this MapScreenState class down here.

Now we can add that build method as always.

And here I now want to build that map.

Now technically, it'll still be a screen though,
and therefore I'll use the good old scaffold widget
also to add a appBar which would hold this back button,
which would allow us to leave that screen.

And here I'll use the built-in AppBar widget
as I always did.

Now in this appBar,

I wanna have a title which should be set based on the condition, whether we're using this screen to show an already existing place or if we're using this map screen to pick a new place. Now to get this information, I can't look at location because this will always be set either to the picked location or the default location. And therefore, I'll indeed add a second property here, a Boolean value, which I'll name `isSelecting`. And I'll set this here as well and I'll set it to a default value of `true`. Because I got a default location in case we don't have a location. And I'll set this to `true` as a default so that if I open this `MapScreen` without overriding this, I am assuming that it was opened to pick a new location. So now down here in the state class, we can use `widget.isSelecting` to either show a text of pick your location. Or if we're not selecting, we could say your location. I also want to add some actions here to the `appBar` or to be precise, I want to add one action that allows the user to save the picked location if we are in this `isSelecting` mode. So here we can use this `if` statement to check if `widget.isSelecting`. And if we are selecting, I want to add one new element to that list, and that would be the only element we might ever have in that list to be precise. And that would be an icon button where I will set `icon` equal to `Icon`

and they are the save icon that's built into Flutter.

And onPressed of course will be set to a function

that in the end should save that place

and return to the previous screen.

And this can be caused here.

Now we'll come back to this function later.

For the moment, I'll instead move on beyond the appBar

and add the body to this scaffold widget.

Because of course this now is the actual map

that should be displayed.

And here we can now use the GoogleMap widget,

which is available

thanks to this Google Maps package which we installed.

Now for that, the import to Google Maps flutter

must be part of this file.

So this import here.

And with that import added,

we can use the GoogleMap widget to render a map.

And this map now of course, also can be configured.

For example, you must set the initial camera position,

which simply defines which part of the map

should be centered.

And here you must create a camera position object

with help of the camera position class

provided by the Google Maps package,

which takes a target parameter,

which is set to a LatLng object with the LatLng class,

which is also provided by that package.

So all these things are available

because of this flutter Google Maps import.

Now here we want a latitude and longitude,

and of course we have that because we have the widget

and there the location.

And there we have a latitude

and we also have a widget location longitude here.

So that's all available.

And therefore we can create this lat long object.

We also should set assume level.

And here I'll set this to 16,

though of course you can play

around with different values here

to, well, show more or less of the map.

Now with that,

besides setting this initial camera position,

we can also add a markers list to the Google Map here.

And of course, as always,

you can press controlled space here

inside of that Google Map parameter list

to learn what can be set there.

You see there are a bunch of parameters that can be set,

but here it's markers,

and markers allows you to create a so-called set of markers.

Now a set is a new data type,

which we haven't used before.

You create it with curly braces like a map,

but unlike a map, it's not a selection of key value pairs,

but instead just a list of values.

So it's like a list

but created with curly braces instead of square brackets.

And the special thing about a set

is that it doesn't allow duplicate values.

If you would try to add an already existing value to a set,

it would not be added, it would not work.

In a list, that's different.

If you have a list of strings,
you can have exactly the same text in that list
as often as you want.
In a set, that does not work.
And markers simply wants such a set of marker objects here.
So a set it is,
and in there we can create a new marker,
which should mark the selected location.
For that I'm creating this new marker object here
with the marker class
that's provided by the Google Maps package.
And here we then should provide a markerId,
which is actually a markerId object created
with markerId with that class here.
And here you should simply provide a unique id.
And since these markers are not dynamically generated here,
but instead only have one marker here,
which I'm creating in code,
I'll give it a hard coded idea of m1.
And this can now be a constant value.
And a marker also needs a position
which defines where the marker will be placed.
And here I'll set this equal to a lat long object
because that is what position wants,
and that's the same object as up here
with widget locational latitude
and widget location longitude like this.
With that, we're showing such a marker
and we don't have the logic for picking a place yet,
but we have the logic for showing a place
and hence that's what we can do now.

24. Displaying the Picked Place on a Dynamic Map

In the `place_detail.dart` file
on this `place_detail` screen,
I want to show this `MapScreen`
if we tap this `CircleAvatar`.
Now for that to make this tapable,
we can't use a built-in parameter
because it doesn't have a `onTap` parameter.
But you, of course, learned before
that we can simply wrap this widget with another widget
with the `GestureDetector` widget, to be precise,
which allows us to add extra event listeners.
So here we can now add `onTap`
to register on taps on that child widget.
And here up on a tap, I, of course,
now want to open that `MapScreen`.
So here we can use `Navigator.of(context)`
to push a new screen onto the stack of screens
using a `MaterialPageRoute`, as always.
And then in the builder, we get our context.
But the screen I wanna return here
now is my `MapScreen` created like this.
And the `MapScreen` here, of course,
is coming from that import here
where I import my `map.dart` file
from the `screens` folder in my project.
So with that added, I'm going to this screen here.
And now I just need to make sure
that here in `place_detail` where I want to view
an already selected place on this map,
I do set the `location` parameter here
to that place location, which I have here,

so that this place is shown on the map
instead of that default dummy place,
which we set up here in the constructor function.
This one, I don't wanna show this one.
I wanna show my own place instead.
And in addition is selecting should be set to false
because we're not selecting a new place here.
Hence, for example, this Save button
should not be shown in the app bar.
And now with that we can save it all, restart.
And if I now go back and create a new place,
we still can't select it on a map.
But I can, of course, take that picture
and then, thereafter, get my current location.
So click this Get Current Location button.
And if we now add this place and we tap it here,
we can also tap this circle avatar.
And we load our full screen movable map here.
You can also pinch and zoom here in the simulator.
On Mac, this can be simulated
by holding the control or command key
and then dragging your mouse.
And this all works.
We also have that pin here.
And we can go back, of course.
And therefore, the next step now
is to make sure that Select on Map works
and allows us to select a place.

25. Handling Map Taps for Selecting a Location Manually

So now to finish this map screen,
I now wanna make sure

that we can also tap on this map to select a new place.

Now for that, we need to listen for taps on the map,
and thankfully that's straightforward,
because the Google Map widget has a on tap parameter,
which can be set to a function
that automatically receives the position
on which the user tapped.

And that position will be such a lat longitude object.

Now here on the map, I then actually wanna move the marker
to that position on which the user tapped.

And I of course also wanna make sure
that if that save button here in the app bar is tapped,
so when this function here is triggered,
that this picked location is sent back
to that add place screen.

Now therefore, I'll first of all add a new variable here
of type lat longitude, which might be null,
which will be my picked location.

And I wanna update this when the map is tapped.

So here on tap, I'll call set state
to also trigger the build method again,
and set picked location equal
to that position I'm getting here.

Now with that we can also move the marker
by going down to that marker.

And here the position should not always be set
to the widget location latitude and longitude.

Instead, I want to check if picked location is not null.

And if that's the case, I want to set my latitude
and longitude to the picked location.

So therefore, here, I'll check if that is not equal to null,
in which case I want to set picked location here

as a value here for position.

Otherwise, I'll use this manually
constructed latitude/longitude value.

Now picked location here could be null.

Dart does not understand that here I'm checking
for it to not be null.

Hence we must add an exclamation mark.

And with that, we implemented this check,
and we have these two alternatives now.

By the way,

you could also write this

in a more concise way by using this syntax,
these double question marks.

This special Dart operator simply says that you want
to use this value here as a value
for this parameter if it's not null,
but if it should be null.

So if picked location should be null here,
you want to use this as a fallback instead.

So that's simply a more concise
and shorter way of getting the same result.

But with that, we're moving the marker,
and I actually don't even want
to show a marker if we don't have a picked location.

So therefore here for markers, we can check
if picked location is null,
in which case I wanna pass an empty set to markers
and only otherwise I want to use this set with my marker.

Though if we would only add this check here,
we would also not show a marker if we're using this map
for showing an already picked place,
because in that case, picked location would be null,

but of course we still would have a place,
but that's why we also have this is-selecting property here.
With that down here,
when setting the markers,
we can check whether picked location
is null, and wrapped in parentheses
to group these two conditions together.
And if widget is selecting is true,
and this is kind of redundant since
is selecting already contains a Boolean.
With that we're saying that we don't want
to show any markers if we don't have a picked location
and we are in this is selecting mode.
If is selecting should be set to false,
so if we're not selecting a new place,
but instead we're using this screen
to show an existing place,
we don't care whether picked location is null or not,
this will never be true,
and therefore we will show a marker,
because we only make it into this part
of the ternary expression
if both conditions here yield true,
since we're using AND for connecting them.
With that,
we're making sure that this map screen stays versatile
and can also be used for showing an already picked place.

26. Using the Map Screen in the "Add Place" Form

So with that, we're managing the markers,
we're able to tap the map,
now we also must pass a selected location

back to the Add Place screen.

And of course, you learned before in the course how you can pass data back from screen B to screen A.

Here, in `onPressed` on this `IconButton`,

we can use `Navigator.of(context)`

and `pop` this screen.

And `pop` allows us to set some data

that will be passed back to the screen we came from.

And here, I'll simply pass back my `_pickedLocation`.

With that, we just have to go to the `add_place.dart` file and wire up these different things.

To be precise, we should go to `LocationInput`,

which is the actual widget

responsible for picking a location.

And in that file, in that widget,

we got this button, this one,

for selecting a location on a map.

And here I now wanna trigger a new method,

a method which I have yet to add.

I'll add it maybe right above `build`.

The exact location does not matter.

And I'll name it `_selectOnMap`,

and it's now this function

that will be connected to this button down here.

And inside of `_selectOnMap`,

we now want to use the `Navigator.of(context)`

to push a new route, the `MaterialPageRoute`,

where we in the end load the `MapScreen`,

which, of course, therefore also must be imported as always.

Now, this `MapScreen` will eventually return the picked place

because we're passing it back here with `pop`.

It will pass back

such a latitude, longitude value to be precise.

Therefore here in location input,

we can add async here.

And with async added,

we can await this to get the pickedLocation,

and we should add this generic type annotation to push

to make Dart aware of the type of value we will get back.

And here this will be set to LatLng object.

And for a lat longitude to be available in this file,

you also must import

the google_maps_flutter.dart file,

this package here,

so this one.

And with that added,

we have the pickedLocation,

which is of this type,

at least it may be of this type

if the user did not navigate back without saving.

So therefore here, we wanna check

if pickedLocation is null,

so if maybe no location was picked,

in which case I'll just return here

and not do anything else.

But if we do have a pickedLocation,

we, of course, wanna use that pickedLocation.

And what does using mean here?

Well, in the end, it means that we again

wanna get the address as we did it here

and then updated our state with the latitude and longitude

and the address.

And therefore we could of course copy all that code,

but to avoid this unnecessary copy step,

I'll instead add a new method here,
which I'll name `_savePlace`,
where I expect to get a latitude parameter of type double
and a longitude parameter of type double.
And it's now inside of `_savePlace` of this newly added method
where I want to get the address and update my state
and also call `onSelectLocation`.
So therefore I'll cut this code
from the `_getCurrentLocation` method
and add it here in `_savePlace` instead.
Of course, here I need to update my injected values
in this string
because now we have latitude and longitude
instead of lat and lng.
Here, we should also add `async`
so that we can use a `weight`
and therefore implicitly `_savePlace` will return a future.
And we can also reflect this here
that we will return a future,
which then won't return any value inside of it though.
And down here we also have latitude
and longitude for creating the `PlaceLocation`.
But with that, we can now call `_savePlace`
from inside `_getCurrentLocation`.
So here at the end I can call `_savePlace`
and pass my lat and lng to `_savePlace`,
so these two values here.
And in `_selectOnMap`,
I also want to call `_savePlace` here.
And now pass the `pickedLocation.latitude`
as a latitude value
and `pickedLocation.longitude` as a longitude value.

And with this change made,
it won't matter how the location is selected.
If we do that automatically
with the user's location or manually on a map,
we will always get this address
with this step here
and then create such a PlaceLocation
and also communicate this to the parent widget,
so to the Add Place screen.
Now, here we can add const,
and here we also don't need to set isSelecting or location
because isSelecting is true by default anyways,
and the default location is fine for me.
And therefore here we should be good to go
and we should now be able to pick a place on a map.
If we go back here
and maybe also reload to make sure
that everything starts from scratch.
And I add a new place,
take my picture, (device beeping)
and then Select on Map, the map opens.
I can tap somewhere and the marker is added there.
And if I then click back without confirming,
I don't see the place here.
But if I do go back and pick my place
and then save,
I go back, but now this place is shown here.
And now if I add this,
this place indeed was saved
and I can also preview it here
without being able to save this place.
I can still tap somewhere

and move the marker though,
and of course that should not be possible.
And to make sure that it's not possible,
back in map.dart,
we can go to this onTap listener
and simply check whether widget.isSelecting == null,
in which case I'll set onTap to null,
which will simply disable this listener.
And I'll only set it to this function
if isSelecting is not null.
And of course here,
I don't want to check for this to be null,
but instead if it's false,
and therefore we can also shorten this
and check for !widget.isSelecting.
This exclamation mark at the beginning means
that we basically invert this Boolean value.
And if this is false,
not is false returns true.
If this would yield true,
not true would yield false.
So therefore now if we are not selecting,
onTap will be set to null.
Otherwise onTap will be set to this function.
And with this change made,
if I now tap here,
I can't move the marker,
and we still don't have the save button,
I can only go back like this.
And with that, we got all the place
related functionality that we wanna have.
Now, the last big feature that I wanna add to this app

is that all that data is saved on the device
so that if I restart the app
because I closed it, for example,
the data is not lost anymore.

27. Installing Packages for Local (On-Device) Data Storage

So now to conclude this section,
I wanna make sure that the data is stored on the device
and therefore not lost if the app is restarted.
Now I wanna do this from inside my provider here
from inside the UserPlacesNotifier class to be precise.
Here we're adding a new place,
and it's here where I then also wanna store my place data
on the disc of this device.

And when it comes to storing the data,
that means that I wanna store the title
and also the location data, the coordinates,
and the human readable address.

But I also wanna store the image
because at the moment I'm using the image,
which is returned by image picker.
And this image picker image that we're getting back,
this picked image is actually stored
in a temporary location.

So it might be deleted from that location
by the mobile phone without us doing anything.
So we can't rely on the image to stay there.
It might be deleted by the operating system
to free up some space.

So, we should actually move that image
to a more persistent location.
And I wanna do that as part of this addPlace method

in this provider class,

in this notifier class to be precise as well.

Now, to store the data on the local device,

we'll need to install a couple of packages,

for example, Flutter path provider,

which is a package that will give us easy access

to the path where we should store the image,

so that it's not going to be deleted

by the operating system.

This package gives us easy access to a path

that we may use to store data

that belongs to our application.

So we will need to install this package here

by running `Flutter pub add path provider`,

which is what I'm doing here.

And once this is installed,

we'll also need to install the Flutter path package,

which might sound similar,

but is a different package

that simply simplifies the process

of working with file paths

because we'll need to construct some paths manually

and this package simply makes it easier to construct paths,

such that they work on different operating systems.

For that, we should also install this package here

by using `Flutter pub add`,

actually this second command here.

So here, I'll run `Flutter pub add path`.

And last but not least, with that installed,

I also want to install the sqflite package,

Flutter sqflite.

And that is a package that will allow us

to store data on the device using SQL commands.

It essentially helps us create a basic SQL database on the device and store data, like the title of a place or the location coordinates in that database.

So that's now not a database running on some server, but a database that runs locally on the device where our app runs.

Now I will say that using SQL lite for storing data on the device is just one option.

You could, for example, also use the shared preferences package.

This is another package that allows you to store data on the device, though it's a bit more basic than the sqflite package.

With shared preferences, you store simple key value pairs.

With sqflite, you have that full database which you can use, which is why I am using this package here.

So we'll also install this here in our Flutter project by running this command.

Now with all these packages installed, we can get started.

28. Storing the Picked Image Locally

And I will get started here by storing the image in a more permanent location.

For that here in addPlace, before creating that newPlace object here,

I'll first of all get the path

where I may store data

such that it's not deleted

by the operating system,

because it turns out

that different operating systems
give you different paths
that belong to your application
and that may be used by your application for storing data.
Now, these paths differ
between the different operating systems,
and that's why we installed the path provider package,
because that package then makes it quite easy
to get the fitting path
for the operating system
on which the app is running.
So that's why I'm importing path provider here
as syspaths,
I'm assigning this alias to the package content
because otherwise some parts of it
would clash with the path package,
which we'll use later as well.
So here I'm using this alias.
And then here in addPlace,
we can get our app directory,
stored in a variable called appDir here in my case,
by using syspaths,
and then there the getApplicationDocumentsDirectory method.
Now, this actually yields a future,
which eventually gives us a directory object.
And hence here I'll add async
and await this.
Now we must copy the image we're getting here
to this new path.
And to do that, we can use this image,
which we get here as a parameter
and called a copy method.

Now, here we must enter the new path
where it should be copied to,
and that of course should be this path,
but copy actually wants a string,
not such a directory object.
And in addition,
and that's very important,
this target path here
must include the target file name as well.
So I'll construct this by, first of all,
getting the file name,
by using the path package,
which we also installed.
So by using package path,
and then there path.dart, like that.
And I'll also assign an alias to that,
the path alias.
And then down there we can use this alias
to access the content of the package.
And there we'll have a basename method,
which can be executed on that image.path
to in the end get the file name
that's included in this overall image path.
So the last segment of that path, so to say.
As you can also see if you hover over basename
and you take a look at the examples.
Here you see that this method
can be used to get the file name.
So now that we have that file name,
we can copy the image to a new path,
which should be a string,
where the first segment

can be taken from this appDir object
by injecting a value where I use appDir.path.
And then we should add a forward slash,
and then specify the file name,
which of course is the file name
I extracted from image.path here.
Now copy then returns a future
because of course this copy operation
can take a while,
and that future then yields the updated file
that was copied to the new path.
Therefore, here I'll await this,
and I'll get my copied image.
And now it's actually this copied image
that should be stored here in this Place object,
because that's now the image that was copied
to the correct path.
So that's now one behind-the-scenes change.
And to see that it works,
I'll stop the running process
and restart the app,
since I installed a couple of new packages
and I wanna make sure that everything works just fine here.
And now with the app restarted,
I'll open the debug console
to see whether I get any errors here.
And if I now add a new place here
and I take a picture for that.
(phone dinging)
That should all work without issues.
And I can see the image here.
So looks like saving it still worked.

And I also see no image-related errors here.

So that's all looking good,
and it looks like copying and saving that image
works just fine.

And with that, we took the first step.

The next step now is to store all the metadata, so to say.

The title, the image path,
and the location in that SQL database.

29. Storing Place Data in a (On-Device) SQL Database

So now to store all that metadata in a database

on the device so that it's still available,

even if the app restarts,

I'll go back here to the `user_places.dart` file
in that provider's folder.

And here after copying the image and maybe
after creating this new place, I now wanna reach out to
that SQL database and store the data in there.

Now to do that, we'll need

to import some extra packages into this file.

We need to import the `sqlite` package.

And then from there, the `sqlite.dart` file
and also another file.

And that would be the `sqlite_api.dart` file.

So `sqlite.dart`

and just `sqlite_api.dart`.

These two files from the `SQL` package must be imported here.

And with them imported, we can now reach out to the database
by assigning an alias to that first import

to the `SQL` of `lite` import of `SQL`

to get easy access under one namespace

to the entire package content

or to the entire content of this file to be precise.

And now we can use this sql object here

to access the package content.

And here, I'll call, `getDatabasePath`, which does

what the name implies, it in the end returns a future

that will eventually yield the path

of the database on the device.

So here, we'll get the `dbPath`

and we can await this of course.

And once we get that path,

we should execute `sqliteDatabase`

to open a database in a given path.

And now it's not this path,

but instead a path we have to create with help

of this path package

because this package gives us a join method,

which can be used to construct a path.

And here, we wanna join this `dbPath`.

So this path we got here with any database name

of our choice

because `dbPath` does not point at a specific database,

but just at a directory where we may create our database.

But to create the database or use an existing database,

we then also must specify the database name

when trying to open it.

And if you try to open a database that doesn't exist yet,

this SQF lite package will automatically create

the database for you.

So that's quite convenient.

So here, I'll name it `places.db`,

and you should end with `.db` here.

And with that, we'll have a database that's either opened

or create it.

Now open database also takes another parameter here, actually a couple of parameters as you see.

But here, it's the onCreate parameter that we should set, onCreate then takes a function as a value.

And this function will be executed when this database is created for the first time.

So if this file in the database path is not found as mentioned, the file will be created for you, but then this function will also be executed and this function is executed to allow you to perform some initial setup work.

Now for that to perform that work, you should actually return a future in your function.

So here, we should return something and that something should in the end be the result of executing a so-called database query, a query that should be executed initially when the database is created to make sure it's configured appropriately.

For that, we can use this db value, which we're getting here, which the end gives us access to the database that was just created.

And on that, we can run the execute command, Q execute a SQL query.

Now this is not a sequel course,

I do have a separate course on that if you would want to learn more about it.

Therefore here, I will just give you the query that should be executed.

And the query is CREATE TABLE user_places.

The table name is up to you, but you should always use the same table

when interacting with it.

And then thereafter, you have opening
and closing parentheses

where we in the end now specify the columns, the fields,
you could say that make up this table.

And here I wanna have an id field, which will be
of type TEXT.

And that's how you write this in SQL
and which will be my PRIMARY KEY.

So in the end, the unique identifier of an entry in
that table and then separated by a comma,

I define my next column, which I'll name, title,
which contains some TEXT, then a comma, image TEXT,
and an image will be TEXT

because I will only store the image path,
not the entire file, just the path to the file instead.

And then for the location, I'll add a field called lat,
which will be of type REAL, which in the end is a number
with decimal values in sql

and then an lng field, which is also REAL
for the latitude and longitude.

And last but not least, for the human readable address,
I'll add the address field, which is of type TEXT.

Now with that, there's just one thing left.

After specifying onCreate here
on this openDatabase function call,
you should also add the version parameter
and set this to one.

And theoretically, you should increase this
whenever you change this query so
that a new database file would be created
if you would change the structure of the database.

Of course, you can learn more about this SQF lite package and its various features on the official package page.

With that though, we're opening the database and we have our initial configuration code in case it should be needed.

And now `openDatabase` will in the end, yield a future, which will then eventually give us that database with which we can work.

So here, I'll get that database by awaiting for `openDatabase`.

And now in the line thereafter, we can finally work with the database, which will have been opened or initialized.

And here I now want to insert my data, my title, and so on into this `user_places` table.

For that, we can use the `insert` method on that DB object that was returned by `openDatabase`.

And now this wants the name of the table where the data should be inserted in this case `_places` and then the different values that should be inserted.

Now the values should be a map where the keys are basically your column names, `id`, `title`, `image`, and so on.

And the values are then, well the values for those columns.

So here, it's a new map where I want to set `id` equal to `newPlace.id`.

And therefore, I guess you should create this object before storing it in a database so that you have this `id` since this is created by the place constructor.

Then for the title column,

I wanna store `newPlace.title`.

And it's really important
that these keys here match the names set up here
in the onCreate function
because we're now assigning the values
to those columns here.
Now for the image as mentioned, it's not the file
but the path of the image.
And therefore here, it's this new place.
And then the image.
And this image file has a path property,
which holds the path, for lat,
it's newPlace.location.latitude.
And for lng, it's newPlace.location.longitude.
And then last but not least, for the address, it's
of course, newPlace.location.address.
And with that, we're storing all that data in the database.

30. Loading Data from the SQL Database

So now that we're storing data
in the database,
we of course also should make sure
that we're loading data from the database
when the app starts.
And for that, I'll add a new method
to the UserPlacesNotifier.
I'll name it loadPlaces,
though the name, of course, is up to you.
And in there, I also wanna reach out to the database,
but then I don't want to insert the place,
but instead get my places from there.
Now therefore, we again need to open the database
as we did it here.

And of course I don't just wanna copy
and use that same code again,
so therefore I will instead cut that code from addPlace
and add it here outside of the UserPlacesNotifier
in that file.

But not like this,
but instead wrapped into a function that can be used,
a function that will return a future,
which will eventually yield a database.

And I'll name it getDatabase
or _getDatabase to be precise
so that it's only usable inside of this file.

And in there, I'll paste in that code.

And in the end here,
return that database that I'm getting here.

So in the end I'm returning db.

And here we should add async so that we can use await.

With that, I'm returning this database
in this utility function,
and now we can use _getDatabase here in addPlace
to get the db here
for awaiting here when we call _getDatabase,
but we can now also get the database here in loadPlaces.

So we're now reusing this functionality,
and here we, of course, also must add async for that.

So with that, we're reusing this logic here.

And now in loadPlaces,
I want to use this db object
and simply call query,
which is a method that queries a database table
and gets the data from there.

And here we must pass in the database table name,

user_places, as specified here.

And you could now actually also specify some conditions.

You could, for example, add a where parameter,

which wants a string

where you could specify some where clause

in case you got some SQL knowledge.

But here I want to get all the places,

and that's exactly what query will do,

it will query and get all the places.

And it will therefore then actually return a future,

which then yields a list of maps.

And this will simply be a list of database rows,

and every map will represent one row in that database.

So therefore here, I'll get my data by awaiting db.query.

And I now want to convert that data to a format

that better suits my needs.

For that, on that data list,

which it is as mentioned,

I'll call map to convert every list item

to a different item.

And then you learned

that here we have to pass a function to map,

which automatically receives that list item,

which in this case represents a row in that database.

So I'll name it row because this name here is up to you.

And then in here in this function,

I wanna return the converted value.

And we can also use this shorter syntax

with this arrow notation here.

Now I wanna convert every row in this database

to a place object using my own place class.

And of course, that place then must be populated

with the title, the image, and so on.

Now, that title can be extracted from the row
by accessing the title key.

Keep in mind that row will be a map
where in the end the keys will simply be the names
of the columns in the database,
and there of course we have that title column,
so that's what I'm accessing here.

Now, here I then also need to tell Dart
which kind of value this will be.

And I can do this by adding the `as` keyword,
which you also saw early in the course already,
which I can use to explicitly assign
or to be precise overwrite
a type that would otherwise be used.

And here, this will be of type string.

For the image, I'll access `row['image']`,
but that would be wrong

because here I actually want to get a file,
image wants a file,
so we should create one by using the built-in File class,
which is available if you add the `dart:io` import.

And then we can create a new file
based on a path which is convenient.

So we can pass `row['image']` to File
because this image here in the database
will hold the path of the image.

And here, we also must add `as String`
to ensure Dart that this will be a string.

And for the location,
we want to create a new `PlaceLocation` here like this.
And of course, in that `PlaceLocation`,

we now also must set latitude, longitude,
and the address.

And the latitude is now row['lat'] as double,
the longitude is row['lng'] as double
using all those column names.

And address, of course,
is row['address'] as string.

And this now creates this new Place object.

Now there's one problem with that,
and that's the ID.

I did store the ID in the database,
but Place actually always creates a new unique ID.

And that's generally good to have,
but of course I do want to have the option
of setting an ID manually.

I do want to accept an ID parameter here,
but now this clashes with this automatic ID generation here.

So what I want to do here
is I wanna accept this id parameter,
but I'll not automatically assign it to this property.

Instead I accept it here.

And then here I'll set id equal to either the id I got here,
or if that's null,

with that special double question mark operator,
I'll set it to the autogenerated id.

That allows me to overwrite this automatically generated id
if I want to,

but it's still the default behavior to get one.

And with that here,

when loading the place from the database,

I can now also set the ID here
and set that to row['id'] like this.

And actually here, we should make it clear
that this will be of type string
or null, of course,
because it may not be set, it's not required.
And hence, here we should also add as String.
So that now transforms all the data.
This transformed data
should be stored in a variable called places.
And I'll also call toList here
to convert this iterable to a list.
And then loadPlaces should set the state
of this notifier class here in which we still are
equal to this loaded list of places.
And now with that done,
we can go back to the places.dart file
in the screens folder.
And here, I actually only want to show this PlacesList widget
if we're done fetching the places from our database.
Even if that should lead to an empty list,
I still only want to show this PlacesList here
if we are done fetching the data.
And this could be achieved with help
of a FutureBuilder here,
a widget which I also mentioned in the previous lectures.

31. Using a FutureBuilder for Loading Data

FutureBuilder is a built-in widget
that takes a future
and then will build a certain widget tree
once that future resolved,
so once that future is done.
And it gives you this extra snapshot here

besides the context in this function,
which it will execute for you,
which simply gives you information
about the current state of the future.
So here for example,
we can then take a look at this snapshot,
and if it's `connectionState` is equal
to `connectionState.waiting`,
I wanna show a loading spinner
in the middle of the screen.
Otherwise I wanna show my `PlacesList`,
so like this.
So if we are still waiting,
I wanna use the center widget instead,
which can be `const`
and center a `CircularProgressIndicator`
in the middle of the screen.
That's what I wanna do if we're still waiting for the data
to be loaded from the database.
Otherwise, once it was loaded,
no matter if it's empty or not,
I wanna show the `PlacesList`.
Now, the future I wanna use here is of course the future
that is yielded by `loadPlaces`.
Since we use `async` here,
this will always return a future,
so we can update this return type
to make this a bit clearer.
And it's this future I wanna wait for,
because it's in here where I'm loading my places.
So to set this future here in `places.dart`,
we should actually convert this to a `StatefulWidget`,

or to a ConsumerStatefulWidget here to be precise,
because we need such a StatefulWidget to be able
to call initState in the state class.

And speaking of that, we should
of course add the _PlacesScreenState class,
which extends the ConsumerState,
which is connected to PlacesScreen, like this.

And here we should close this class and add createState,
where we return such a ConsumerState related to PlacesScreen
and where we return a new instance of _PlacesScreenState.

Now build no longer gets this ref,
instead ref is now a generally available property here.

And now with that here we can add initState
and set up our future,
the future that will be connected to this FutureBuilder.

So for that here I'll add the _placesFuture property,
which will be of type Future
that will actually not yield any value once it's done.

And I'll decorate this or annotate this
with the late keyword, which is a special Dart keyword
that in the end tell Dart that this is not set initially
when there class is instantiated,
but it will be set before it's really needed,
because we will indeed set it here in initState.

Here, I'll set _placesFuture ==
using the ref keyword
and reading from the userPlacesProvider
and there from the notifier,
the result of calling loadPlaces.

This will then, behind the scenes,
load the places from the database,
because that's this method

and update the state of the provider,
which is important for the build method thereafter,
because here I'm watching those places,
but I will load the places from inside initState
and store the future that's returned by loadPlaces,
in _placesFuture.

And this now therefore does not just kick off
that loading process,
but it also allows me to use _placesFuture
in my FutureBuilder so that this FutureBuilder makes sure
that we either see that loading spinner or the PlacesList.

And with all that done, if you save everything
and reload the app, you should see, No places yet.
But you also shouldn't see any errors here.

And also not in the debug console
if you take a look at that.

And if I now do add a new place here, like my office,
where I take a picture, (phone chimes)
get the current location and then choose Add Place,
this works, still no errors here,
and I'll also add a second place,
just to be sure that it really works.

(phone chimes)

And pick this one on the map,
save this.

Now the data is there.

I can of course also view these places, so that also works.

But now if I reload the app,
or even restart it to be super safe,
this also restarts and my places are still there
and I can still view them.

So this still works.

And with that, we're now storing the data on the device.

And of course this also all works on the iPhone.

There, we can also add new places by taking images

and then getting the current location

or also by creating a new place

and there selecting a place on the map.

And there on the iPhone,

you can also see that you cannot just visit those places,

but that if you close the app

and you restarted in the end, which I did here,

the places are still there.

32. Module Summary

And that's it for this section.

This was obviously another very long section,

but you learned a lot about Flutter

and about how to work with native device features.

We did not just practice some Flutter basics,

like working with widgets or adding such a provider,

but most importantly, you for example,

learned how to pick an image

with help of the image picker library,

or how to work with locations

either with help of the location library

by getting the current location of a user,

after checking that we can do that.

Or by using Google Maps

through the Flutter Google Maps library

to show this map screen where we were able to tap on the map

and pick a location by doing so.

So you therefore, of course, also learned in this section

that especially when working with native device features,

you often have to use third party packages.

And I also mentioned that of course, you should always go through the installation instructions of those packages to make sure that they work fine.

Now, then in this section, we of course also worked with the Google Maps API to convert a location to a human readable address or show such a snapshot review.

And we of course also made sure that data is stored on the device, that the image is copied to a safe directory, but also that we have a database with which we can work, a database from which we can load places, but of course also a database to which we can save places.

And no matter what you're going to build, chances are high that at some point you will need to use some native device feature in your app.

And maybe this section already covered the features you might need in your app, but even if it did not, you now should have a solid understanding of how you can use native device features in Flutter, and that third party packages are typically the way to go.

33. Adding Your Own Native Code

With Flutter, when using native device features, you'll very likely use external packages most of the time - like the ImagePicker or GoogleMaps packages used in this section.

But if you need to tap into a certain native device functionality that's not exposed by any package, you can also write your own native code and connect that to your Flutter code.

It's a rarely needed feature but this page from the official documentation explains how you could add your own native code: <https://docs.flutter.dev/development/platform-integration/platform-channels>

13_push-notification

1. Module Introduction

This course section
will be another pretty big course section
because in this course section here,
we're going to build a chat app,
and therefore, of course,
an app where users can send and receive chat messages,
but also an app that features push notifications
that are shown whenever a message is received
when the user is currently not using the app.
In addition, this app forces users to create accounts
and log in before they can use the app.
And as part of the signup process,
users also have to take and upload an image.
And therefore, this app,
which we're going to build in this course section here,
has multiple advanced features,
which we haven't used up to this point.
Like for example,
user authentication, image upload,
and push notifications,
but it will also provide plenty of opportunities
to practice what you already learned.
Because, of course,
we're also going to build a bunch of widgets
and add a bunch of styling and do all those things,
hence using all the knowledge you gained
throughout this course.

You will also learn how you can connect a backend via a SDK instead of sending HTTP requests yourselves.

And I'll get back to this point and what it actually means a little bit later in this section.

But then as mentioned, we're also going to add user authentication, image upload, push notifications, and much more to this application here.

2. App & Firebase Setup

Now for building this application here, I created another new Flutter project. I named it CHAT_APP, and I replaced the default main.dart file with the one you also find attached, which, in the end, just defines a very basic app widget that adds some themeing to the app.

But we're going to build all the rest together from the ground up, step by step, and along the way, we'll, of course, practice what you already learned throughout this course, but we're also going to add all these new advanced features, like image upload, authentication, and push notifications.

So let's dive right in.

And in this app here, which we're about to build, all users must create an account and sign in before they can start sending or receiving messages.

So that will be the first step.

Adding user authentication will be the first step,
and therefore, of course,
we need a screen and a form,
so a combination of inputs,
that, in the end, allows users to sign in
because when we talk about user authentication,
we're, in the end, talking about the frontend
and the backend communicating with each other again.
The frontend is our Flutter app,
and the backend could be any remote server,
either a server fully controlled
and programmed by you with your own server-side code,
for example, written with Node.js or Python,
or a third-party service, like Firebase.
Either way, as part of that sign up and sign in process,
your frontend app has to collect the user credentials,
the user data, so email and password, for example,
and it then must send that data to the backend
where it can be stored in a database,
but where it also can be validated
so that after you created a user,
once the user wants to log in,
those credentials are sent to the backend again,
and then there, the backend basically checks
whether email and password
match the email and password stored in the database.
That's, in a nutshell, what authentication means.
Now, there is a bit more that belongs to it,
and I'll get back to that,
but since it all starts
with collecting these user credentials

and sending them to a backend,
we need these two things to get started.

We need a screen

and a form in the Flutter app
that does allow a user to enter their credentials,
so email and password, for example,
and we need a backend
that supports user authentication.

Now again, you could be writing such a backend on your own,
and I have courses on that if you're interested.

I have a Node.js course, for example,

where you do learn

how to write server-side code with JavaScript,
but this course here is, of course, about Flutter,
and Flutter is for building mobile apps,
for building frontends,

and therefore, here, we're again going to use Firebase
as a dummy third-party service because Firebase,
which we already used in the HTTP section
earlier in the course,

does not just offer this real-time database feature
that we use there,

but instead, also supports user authentication
and the storage and validation of user credentials.

Now, for that, you must log into your Google account,
and then either create a new Firebase project
or use your existing project.

And here, I'm creating a new Firebase project,
which I named flutter-chat-app here,
but the name is up to you.

We don't need Google Analytics.

And with that, we can create this project,

and once that project was created,
you can go to the Build section here again,
and there, you'll find this Authentication area.
And it's there
where you can enable authentication with Firebase.
For that here, we can click on Get started,
and then here choose Email/Password
because for this app,
I want to allow users to log in with email and password.
Now, if you should use Firebase
for future Flutter project you might be working on,
you also see here that you have more options
than creating and managing users with Email/Password.
You could, for example, also allow users to log in
with their Google, Apple,
or a couple of other accounts.
That all works,
and the official Firebase documentation
is the place to go if you plan on using Firebase
in your next Flutter projects.
For this course,
where I'm only using it as a dummy backend, basically,
I'll choose Email/Password here
so that we can create new users
with an Email/Password combination,
and we then also can log users in
with such a Email/Password combination.
For that, here I'll choose Enable,
and then click Save.
And with that said,
you can use Firebase for authenticating users.
That being said, though,

we'll only do that a little bit later
because at the moment,
we have nothing on the frontend
that would allow users to enter their credentials,
and that's, therefore, what we're going to get started with.
We're going to come back to Firebase later
and to sending requests to Firebase later.
For now, we're going to focus
on adding an authentication screen to this app here
so that we can collect
and send those user credentials.

3. Adding an Authentication Screen

So now to build this auth screen,
here in my Flutter project,
I'll add a new folder in the lib folder,
which I'll name screens to store my screen widgets in there,
and in that folder, I'll add a new auth.dart file.
The name of course is up to you,
but in there, I wanna create that auth screen widget,
which in the end will be responsible
for collecting that user input.
Now for that, in there,
we of course create a new class which I'll name AuthScreen,
and which here, extends StatefulWidget,
and I'll extend StatefulWidget here,
because I will render a form
and collect some user data through that form,
and for that, we'll have to use some features
that require a StatefulWidget.
Now, in this StatefulWidget here,
I'll add my constructor, AuthScreen,

where I accept and forward that key,
and where I add that createState method.
And then we also add this _AuthScreenState class
in addition to the AuthScreen widget class
where we extend the State object that will be connected
to the AuthScreen class,
and that will also be the case here, of course.
And in that State class here,
we have that build method as we always do,
and as we always did throughout this entire course.
With that, we can then instantiate
this _AuthScreenState class here in the createState method.
And then here in the build method,
we return our widget tree for this screen.
And here, I wanna start with a Scaffold,
because of course, I want to make sure
that we do have this nice styling
and this background color that we might want to have.
And on that scaffold here,
I'll add the backgroundColor parameter to add,
guess what, a background color to that scaffold.
And here, I'll use my Theme.of constructor function
with the context to get hold of my theme setup
or to my theme settings,
and then there, we can access the colorScheme property.
And then there, I wanna use the primary property,
so the primary color that I did in the end set up.
In addition to setting that backgroundColor here,
I'll also add the body parameter,
and then here, I wanna render the actual auth form.
Now, for that here, I'll render a Center widget
so that all the content inside

of it will take the entire width and height of the screen
and will be centered in that width and height.
And then inside of Center, I wanna have a child
that is a `SingleChildScrollView`
because I'm going to add a column
with multiple text inputs here,
and I wanna make sure that even if the keyboard is open,
all these inputs and buttons are always reachable.
So the therefore, I ensure
that the content will be scrollable
by wrapping `SingleChildScrollView` around a `Column`,
and that `Column` will then hold the actual inputs and so on.
Now on that column, I'll set `mainAxisAlignment`
to `MainAxisAlignment.center`
to again center all the content inside of this column.
And then here, we'll have a couple of children,
of course, a couple of child widgets.
Now, when it comes to those child widgets,
I wanna start with a container
to add some margin to it
with `EdgeInsets`
only to target the different directions individually,
to add a top margin of 30, a bottom margin of 20,
and also a left and right margin of 20.
Those values should ensure
that this container will be positioned
exactly the way I want to position it.
But what's the purpose of this container, you might wonder.
Well, the purpose is to, in the end, display an image,
so that will be the child of the container.
I'll also add a width to the container to set this
to 200 pixels to size this image appropriately.

Now, when it comes to that image,
you'll find the Chat png file attached,
and I will create a new folder in my project here,
which I'll name assets.
And in there, I'll add an images folder,
and in there, I'll store that attached chat.png file,
so that's inside of assets/images.
With that image added here, I'll also go to pubspec.yaml
because as you learned earlier in the course,
if you have some assets like an image in your project
that should be usable in your app code,
you also have to go to the pubspec.yaml file
and comment in this assets part here,
and then simply reference that asset,
so that image, for example, that should be available.
So here, the assets/images/chat.png path,
pointing at the image we added here.
With that added, we can go back to auth.dart,
and in that widget here, inside of this container,
I'll now use the Image.asset constructor function
to reference that image I just added,
which is in that assets/images/chat.png path here.
So with that, I'm effectively setting up a container
with some margin and with a fixed width
that will display this image.
And I'm simply adding this for styling reasons,
to make this app or this screen look good.
Below this container,
I then wanna have the actual authentication form.
So below this container here, I'll add a Card widget
like this, to have this nice card styling,
this nice card look.

And here on this Card, I'll also add some margin
and set this to `EdgeInsets.all(20)` like this
to add a margin of 20 in all directions around the card.

And then the child of this card here will be
yet another `SingleChildScrollView` to ensure
that the content inside of this card is scrollable
and all the content is always reachable.

And here, I'll wrap this around a `Padding` widget
to add some inner padding inside of that card,
so between the borders of that card,
the edges of the card, and the actual card content.

And the amount of padding is then set up
with the padding parameter,
and here, I'll set this to `EdgeInsets.all(16)`.

And then the child of that padding will be that form,
so ultimately that authentication form
which you wanna build,
and for that, I'll use the `Form` widget
provided by Flutter, about which you also learned
in greater detail in an earlier course section.

And this `Form` will then display a `Column` widget,
it will contain a `Column` widget,
which in the end will display the individual form inputs,
so the different text fields that will be displayed.

Now for that here, on that column,
I'll set the main axis size
to `MainAxisSize.min` to make sure
that this column will only take as much space as needed
by its content, essentially, instead of taking as much space
as it can get and having no boundaries on the vertical axis.

And then the children of this inner column in that form,
in that card, will, as mentioned, be those input elements.

And for that here, we can for example,
add a `TextFormField`, which should allow users
to enter their email address.

And therefore, I wanna add a label that tells them such
by adding the decoration parameter
and adding an `InputDecoration` object here
to configure the decoration of this `TextFormField`.

And then here, we can set the `labelText` parameter
in that `InputDecoration` constructor function here
to `Email Address`, for example,
and this can be a `const` object.

Now, in addition to setting that label,
I also wanna set the `keyboardType` of this `TextFormField`
to `TextInputType.emailAddress`,
which as the name suggests in the end, ensures
that the soft keyboard that will be opened will be optimized
for entering email addresses, by for example,
ensuring that the `@` symbol is easy to reach.

In addition, I'll add another property
which we haven't used before in the course,
and that will be the `autocorrect` property,
which I'll set to `false` to turn off auto-correction,
because that can of course be super annoying
if you're entering an email address
and you're getting corrected all the time.

In addition, we can also set `TextCapitalization`
to `TextCapitalization.none`
to ensure that the email address won't get capitalized,
so that the first character
of the email address won't be uppercase,
because that can also be very annoying.

Now with that, we have a basic `TextFormField`,

and I'm not done with it yet.

There are still a couple of key features missing,
but I wanna see something on the screen,
and therefore, I will just copy this
to also add another TextFormField that allows users
to enter their password by adding a labelText
of Password to that next TextFormField.

And here, the keyboardType will actually be
the regular text keyboardType,
and therefore, I'll remove this.

I also will remove autocorrect and textCapitalization,
but here, since this input will be all about users entering
their password, I'll set a never,
very important parameter, on TextFormField.

I'll set the obscureText parameter to true,
which simply hides the characters as they are being entered,
so that if someone were to stand
behind the person using that app,
that person couldn't see the entered password.

Well, and with that, as mentioned,
we're absolutely not done with this form.

There's a lot missing here,
but we have these two input fields in that form,
in that card, on that AuthScreen,
and I just wanna make sure
that we're going into the right direction.

Therefore, I'll save that file, and back in main.dart,
we can now import and use that AuthScreen.

So here, I'll import from my chat_app project,
and there from the screens folder, the auth.dart file.

And with this import added here, we can then
of course go down here, and set AuthScreen as the widget

that should be displayed in our app here,
and here, we can add const.
With that, if we then run this app here,
we will be greeted by this basic authentication screen,
as mentioned, not finished, but not looking too bad.
And here, if I tap into this email address field,
you'll see that a soft keyboard opened up
that makes entering email addresses quite convenient
by giving us this at symbol here, so that we can enter
our email addresses here with great ease.
And the password field,
on the upper hand, gives us a regular keyboard,
but if I enter something here,
you see that those characters are obscured,
which of course also is great.

4. Adding Buttons & Modes to the Authentication Screen

So let's now continue work on
that auth form here.
Back in auth.dart, we should now add some buttons
to that form and then also fine tune those text form fields.
But I'll start with the buttons.
For that here, I'll first add a sized box
to add some spacing between those inputs and the buttons
where I simply set a height of let's say 12.
And then below that sized box here,
I wanna have two buttons, which are not next to each other
but above each other instead though.
The first button will be the main button
that can be used for submitting the form.
And here I'll use an elevated button
where the child will be a text widget

where I simply wanna say "Signup."

And this can be const.

And onPressed then will be a function,

which at the moment will be empty,

but which later will make sure

that all the input values are collected

and sent to Firebase.

That's the idea behind this button.

The second button, which I wanna add below

this elevated button will be a text button here for me,

which the end should allow users to toggle

and switch between sign up and log in mode

because of course this form should be usable

to either create a new account if you don't have one

or log in with an existing account if you do have one.

And at the moment we only have email

and password inputs here,

but later we'll also add image upload.

But the image upload should only be enforced and available

if you're creating a new account.

If you're logging into an existing account,

you already have an image,

and you don't need to upload one again.

Therefore, it will be important to differentiate

between login and signup mode.

And this button here will allow users

to switch between those modes.

So therefore here we of course also have onPressed

and at the moment I'll set this to an empty function,

but we'll also have a child which displays a text widget

where we in the end say create an account

or alternatively log in with an existing account

or something like this

depending on the mode in which we currently are.

And in the end, that's also one reason

for using a stateful widget here

because now we can manage that mode as state in that widget.

Here we can add a new variable

that could be named `_isLogin`

controlling whether we are in login mode or not.

And we could initially set this to true,

so that login mode is our default.

And then down there at the buttons,

we could say that this button, which will allow us to switch

between those modes, either says, "Create an account"

or says "I already have an account."

"Login," something like this.

And here I wanna use a ternary expression

based on the `isLogin` variable we just added,

the `isLogin` property,

and therefore here I'm checking whether `isLogin` is true.

If that's the case, this button should allow us

to switch to signup mode

and will therefore say, "Create an account."

Alternatively, it will say,

"I already have an account."

"Login instead," and that's a bit long.

So therefore I'll just say, "I already have an account."

And here in this function I then of course wanna

switch my mode.

So here we can use `setState`

to set `isLogin` equal to true if it's currently false

or false if it's currently true.

And we could write it like this

and use a ternary expression here
and check if it's true, in which case
we set isLogin to false, and otherwise we set it to true.

But that's unnecessarily long.

A shorter way of setting isLogin to its opposite
is to use an exclamation mark here and then isLogin
because this exclamation mark
basically checks for the opposite
as you learned earlier in the course.

If this is true, exclamation mark true will yield false.

If isLogin is false,
exclamation mark isLogin will yield true.

That's the idea here.

So this allows us to easily switch
between these two different possible values.

And this button's text will then be updated
based on the value.

But it's not just the text of this button here,
it's also the text of this button that should be updated.

This should not be const.

Instead here I wanna say Signup if we are in signup mode,
but I want to say Login if we are in login mode.

So "Login" is my alternative text
and I'll check this isLogin property
with another ternary expression here.

And then later this will also influence
whether we're sending a request to Firebase
to create a new user or to login an existing user.

But we'll get back to that later.

So for the moment it's just about those button texts.

And if you now save this, we should see these buttons here.

And if I tap create an account here,

you see the button texts change.

If I create an account, we switch to signup mode.

If I click on "I already have an account,"

we switch to login mode instead.

Now here, I just also want to adjust the button styling,

and I could do this through my theme,

but since in this app here,

this will basically be the only place

where I'll use an elevated button.

I'll adjust the styling right here on the button

by setting style equal to `elevatedButton.styleFrom`

as you also learned a bit earlier in the course

or I guess way earlier in the course,

but either way.

Here we can then set the foreground color

and the background color.

And I wanna set the background color here,

and I'll set the background color

to theme of context color scheme to access the colors

of my globally available theme.

And here I'll choose primary container.

And with that I'll have this nice background here,

but with that we now added those buttons,

therefore it's now time to come back to those inputs

and make sure that those inputs get validated.

And we also then are able to extract the entered values.

5. Validating User Input

So, let's get back to those text form fields.

And here, I wanna do two things.

I wanna add validation

and I want to be able to get hold of the entered values.

Now, I'll start with validation.

And as you learned earlier in the course,
in the forms and handling user input section,
you can add this validator parameter
to TextFormField widgets,
and this then takes a function
which will automatically give you
the currently entered value,
which will be a string or null.

And then in there, you can return an error message,
so you can return the string
if you consider the value to be invalid.

And otherwise, you can return null
if the value should be treated as valid.

So, therefore, here we can use a if statement
to check if the value is equal to null maybe
or if the value is empty, so if it's an empty string.

You also might want to call trim() on the value
before calling isEmpty to make sure
that excess white space is removed.

And here, since this is the email address input,
I also wanna add at least a basic check that ensures
that a valid email address was entered.

Now, you could get more sophisticated here,
but here, I'll just check if not
the value contains an @ symbol, so the @ character.
If the entered text does not contain such a @ symbol,
I consider it to be invalid.

So, if either of these conditions here is met,
and one of them is enough because of this OR operator
that connects the conditions, I'll return an error message,
"Please enter a valid email address."

Like this.

If we don't make it into this if check here,
we know that the value is valid,
and therefore then we should return null.

So, that's my validation logic
for the email address input field.

Next up, let's work on the password field.

And here, I'll also add this validator parameter
and add this function that gets a value.

And here, I will copy this if statement
from the email address input.

And I still wanna check whether value is null,
but in addition, I'll also trim the value
and then check its length.

And I wanna make sure that the length
is at least six characters long,
so if it's less than six characters long,
I wanna make it into this if statement
and return an error message

because the password should have a certain minimum length,
which in my case here, will be six,
also because that's the minimum length
that will be enforced by Firebase later.

So, here, I'll then say,

"Password must be at least 6 characters long."

And if we don't make it into this if check
and the value is valid, I'll return null here.

So, that's the validation.

Of course, as mentioned,

the goal also is to get hold of these entered values,
and for that, I'll add a brand new method
to this AuthScreenState class here.

I'll name it `_submit()`,
and it will not return anything.
And this should be triggered
whenever this `ElevatedButton` is pressed,
so I'll set `submit` here, `_submit` as a value
for `onPressed` on that `ElevatedButton`.
Now, in the end here in this `submit()` method,
I then wanna trigger my validators
and make sure that all the inputs are saved
and I get hold of the entered values.
And you learned that for that,
you can essentially create a `_formKey`,
which, in the end, is a `GlobalKey`
which will be connected to the `FormState`.
And you learned about this `GlobalKey`, this `_formKey` here,
in the forms section earlier in the course.
This global `_formKey` here
can and should then be assigned as a value
to the `key` parameter on the `Form` widget.
And it's then through that `_formKey`
that we get access to the form.
So, here in `_submit()`, we can use our `_formKey` here
to get hold of the current state,
which, in the end, will be our `FormState`
and call `validate` for example.
And actually, I'll just name it `_form`
since this name makes a bit more sense to me than `_formKey`.
But of course, the name in the end does not matter.
But with that we got this `_form` property
that holds the `GlobalKey` and is connected to the form
and we can then call `validate()` on that.
Now, I'll need to add an exclamation mark here, though,

to tell Dart that this will be set.

It will not be null

because we know that `_submit()` can only be triggered via this `ElevatedButton`.

And if that `ElevatedButton` is available,

the form will be available as well,

so it will not be null.

And of course, validate, as you learned earlier,

then returns a Boolean, which, in the end,

tells us whether the entered values are valid or not.

Hence, I'll store that result

of running my validators in a `isValid` variable,

a final variable which we can then use thereafter

in that `_submit()` method

because I can then check if `isValid`.

Alternatively, we could, of course,

also just grab that code and use that a condition here.

But here, I'll do it in two steps

to keep the code slightly more structured and organized.

And if my inputs are valid, I again wanna reach out

to my `_form.currentState`, and I wanna call `save()`.

And you might recall that `save()`

then triggers a special function

that can be assigned to all those text form fields,

the `onSaved` function,

so a function you assign to the `onSaved` parameter.

Now, this function then gets

the currently entered value automatically,

and you can then do with that value whatever you want.

For example, store it in a property of this class,

and that's what I'll do here.

I'll add a new variable, `_enteredEmail`,

which initially is an empty string,
and another new variable `_enteredPassword`,
which also initially is an empty string.
And I then wanna set `_enteredEmail` in the `onSaved` function
of the email input,
so I'll set `_enteredEmail = value`.
And of course, value theoretically could be null here,
though, we of course know that here it won't be
because I'm checking for value being equal to null here
in my validator.
And since I only call `save()` if all my validators passed,
I know that at this point, the entered value won't be null.
Hence here, we can add an exclamation mark
and then store our `_enteredEmail`.
We don't need to use `setState()` here
because here, I actually don't want to update the UI,
hence I'll just store it like this.
And then in the end,
repeat this step here on the password input here.
So, there, I then wanna update my `_enteredPassword`.
And of course, you learned about all that before.
This is just some repetition in the end.
And therefore here, after calling `save()`,
we should be able to print the `_enteredEmail`
and of course also the `_enteredPassword`.
And later, we'll send that to Firebase
instead of printing it.
But for the moment, if I save this, reload everything,
open my Debug Console,
and I then go back here and I enter some email address
and also a password, and I click Login,
nothing happens here

because we haven't added any logic to let anything happen.

But if we go back here,

we can see those input values here.

So, collecting them works and validating them also works

because if, for example, remove the @ symbol here,

you see I get this error message.

If my password is too short,

I get an error message there as well.

So, that all works as expected.

With that, we're collecting the inputs,

and now we can send them to Firebase.

6. Firebase CLI & SDK Setup 1/2

So we're able to collect

those user credentials,

the email address, and the password.

Now we wanna send that data to Firebase,

and we can send it to Firebase

through Firebase official Authentication REST API,

so in the end,

by using certain URLs exposed by Firebase

that are meant to be used to send such credentials

and to ensure that Firebase stores such credentials.

We could send HTTP requests,

as you learned it earlier in the course,

to Firebase to specific URLs defined by Firebase

with specific data attached to it

to create users or to sign users in.

That all works and could be done,

but I'm not going to send HTTP requests like this.

I am still going to send HTTP requests

because the communication between front end

and backend always happens through HTTP in the end.

But I'm not going to create and send

and handle those HTTP requests manually.

But instead, in this section here,

we're going to use the Firebase SDK for that,

and that, of course, brings up one important question,

what is such an SDK,

and why would we use it?

In the end, SDK is a Software Development Kit,

and in the end,

it's basically, just again,

a collection of code you can use in your code,

just as Flutter, in the end,

is a collection of code you can use

in your Dart projects to,

in Firebase case, talk to the Firebase services,

and you would use such an SDK to make your life

as a developer a bit easier

because configuring, sending,

and handling HTTP requests manually can be cumbersome,

especially when working

with a complex service like Firebase.

And that's why many third-party services, like Firebase,

do offer such SDKs,

such prebuilt code packages,

that simplify the process of working with them.

As mentioned, you don't have to use that SDK,

you could use Firebase REST API

to send HTTP requests to certain URLs manually.

And if you, for example, did build your own backend,

you also wouldn't have such an SDK

unless you build one yourself.

But for many third-party services, like Firebase,
such SDKs exist
and being able to use them
and making your life as a developer easier
is something you should be able to do,
and that's why we're going to use
that Firebase SDK in this course section here.

Also because we will not just be using Firebase
for authentication
but later also for uploading and storing images
and for sending push notifications and more.
Now you can simply search for Firebase Flutter SDK
to learn more about how to add this SDK
to your Flutter project.

And at the moment,
all we have to do to use this SDK
is to first install the Firebase CLI,
the Firebase Command Line Interface,
which is an extra tool you can install on your system,
no matter which operating system you're using,
to interact with Firebase
and to, for example,
help with that SDK installation process.

Though, it's worth noting that that CLI is not the SDK.
Instead, it's another extra tool provided by Firebase
to make working with Firebase features,
like that SDK,
a bit easier on your system.

And here I'll go to macOS,
but, of course, you should choose your platform.
And here, for me,
I'm going to install that SDK

by using npm since I have Node.js installed on my system.

Though, you can, of course, use any of these options to make sure that you do install that Firebase CLI.

Ultimately, it will just be important

that, in the end,

you will have that CLI installed.

So here, I'll install it by running this command.

But this will only work if you have Node.js on your system.

And once you did finish

that Firebase CLI installation process,

which you can confirm

by running firebase in your System Terminal

or Command Prompt,

and if you then don't get an error

but instead some output that looks like this,

you will know that Firebase was installed.

So once you did do that,

once you did install this Firebase CLI,

we can proceed with the second step here

on that Firebase SDK page now again

and log into our Firebase,

so in the end, into our Google account here,

by running firebase login.

Now this might prompt you

for your email address and password,

allowing you to log into your Google account

through that CLI.

And once you did that,

we can run this command here

to install the FlutterFire CLI,

which is yet another extra tool provided by a Firebase,

which will specifically be used

for managing Flutter projects with Firebase
and for adding Firebase features to Flutter projects.
But since this FlutterFire CLI
uses the Firebase CLI under the hood,
we need to install both,
the Firebase CLI and then the FlutterFire CLI,
by running this command.
And I know that's a lot of tools
and a lot of CLIs,
but you only need to do this once ever on your system,
and thereafter, you will be able to use Firebase
and this FlutterFire CLI in any Flutter project,
not just in the one we're currently working on.
So here, I'll now run this command
to install this FlutterFire CLI on my system.
And this can, of course, all take a while,
but once this is done,
we can go back to our Flutter project finally
and run this command in that project
using that FlutterFire CLI that we did just install
to configure this Flutter project for using Firebase.
Now if you're getting an error that, in the end,
tells you that the flutterfire command wasn't found,
you might wanna run that command
that, in the end, installed the FlutterFire CLI again
because in there,
you should then see a warning that, in the end,
tells you that right now,
you might have an issue with your path set up,
hence, basically, leading
to the flutterfire command not being available.
And it should also show you a fix for that

so that, for example, here on my system,
which is a macOS system,
I should add this line here
to my terminal configuration file,
which here on my system on macOS, in the end,
is this zshrc file,
which I'll just open with TextEdit.
And then in there,
I'll add this line that was shown to me in the terminal.
Again, you should see the specific instructions you need
for your system in your Terminal
or Command Prompt when running
that `dart pub global activate flutterfire cli` command.
With that fixed here,
if I now close this terminal
and then reopen it,
effectively opening a new one,
I should now be able to run `flutterfire configure`
in my Flutter project here.
And this now, in the end,
walks us through the setup process
for this Flutter project to use Firebase in there
and to add that Firebase SDK.
And again, you only need to go through all these steps once,
which is why it takes some time now,
but it won't take that long for future Flutter projects.
You will always have to run `flutterfire configure`
when you're in a new Flutter project
and you want to add the Firebase SDK to it,
but you won't have to go through
all these general setup steps again.

7. Firebase CLI & SDK Setup 2/2

So here, I'll now pick
that one Firebase project that I created before
and connect it to this Flutter project here in the end.
Then we can choose which platforms we wanna support.
And for that, here I'll disable macos and web
by selecting them with the keys
and pressing space to select or unselect,
so that I only add android and ios
as target platforms.
And with that, this flutterfire CLI will actually change
a couple of configuration files
in this Flutter project
to enable Firebase
to add the Firebase SDK in the end,
and to also make sure
that these platforms are supported
by our Firebase project.
So it will not just change configuration files here
in our Flutter project,
but also adjust some settings
in our Firebase project.
So here I'll confirm that it performs these changes.
And with all of that out of the way,
it will now have made some adjustments
to the ios and Android folders,
hence unlocking certain Firebase features there.
And with that done,
with that flutterfire config command out of the way,
we can now finally start adding
some specific Firebase SDK features
to our Flutter project,

because what we did thus far
was the general initialization,
now we should also add the features
that we wanna use here.

And for that, you can use
the good old flutter pub add command again,
as we used it before in the course,
to add third-party packages,
for example, to add the firebase_core package,
which we must add to use certain Firebase features here.

So that's one package
that we should install into this Flutter project.

And in addition, to log our users in,
we should also run flutter pub add firebase_auth.

That's another package
provided by a Firebase,
that's in the end part
of the overall Firebase SDK,
that will add auth-specific functionality that we can use.

Well, and with all of that,
finally we now added all the packages we need for now,
we made sure that our project
is also configured to support Firebase.

Now we should just again run flutterfire configure
to make sure that we have the latest configuration
we need for the specific packages we installed.

So I'll here go through that step again.

And I'm doing that because essentially
that is recommended here in the official documentation.

And once that's done,
we can set up Firebase in this project
so that we can then use the features

provided by the Firebase SDK in our code.

And to set up everything,
we should continue with the next step
in the official documentation.

We should add these two import statements here
to our main.dart file in the lib folder.

Hence I'm going to do that here,
and add them here.

And with that added,
you should also copy this code here
and add that in your main function,
for which you also should convert this to an async function
to, in the end, initialize Firebase.

You also might notice
that this firebase_option.dart file
was added automatically.

That was done by flutterfire configure.

And it's that file
which we're also importing here into main.dart.

And therefore it's the options set up there,
which were set up automatically by Firebase,
by this flutterfire CLI,
and which you therefore shouldn't change
unless you know what you're doing.

But by importing that,
by adding that and importing that
and by also running this code,
finally everything will be set up
and this app will be ready to use Firebase.

And again, this initial setup process
of course does take some time,
but with that, we're now ready

to use Firebase features in this app.

Now, to confirm that everything still works,

I'll shut down my running app on the emulator

and restart it from scratch.

And now this assembly and this compilation

and creation of this app package

can indeed take some time.

It will not take that long every time you start it,

but if you just added Firebase,

the initial compilation,

the initial assembly can take a bit longer than normally,

but ultimately it should finish,

even if this process should take a couple of minutes,

which is possible.

And it should then run this app

on the emulator again.

And if you then should get stuck on this starting screen,

you also should add another line of code

in front of this initialized app code here.

And you should call `WidgetsFlutterBinding`,

`ensureInitialized`,

and then restart this.

And you should see this auth screen again.

And with that, we have the app as we had it before,

but now with Firebase added to it.

And of course, since this overall setup process

took some time

and we had to go through a couple of steps,

even though those at least

were well documented

and therefore not too hard to replicate,

but still, since we had to go through all these steps,

it might not yet make a lot of sense
to add Firebase,
and sending and handling HTTP requests manually
might still seem more attractive.
But as you will see throughout the rest of this section,
now that we did add Firebase,
it will be super easy
to tap into all these
various Firebase services and features,
and adding authentication, image upload
and later also push notifications,
will be much easier
when using the Firebase SDK
than if you have to do it all manually
with the Firebase REST APIs.

8. Signing Users Up

So now that we finally installed
the Firebase SDK and got everything set-up there,
we can go back to our off-screen
and there to the submit method to finally create a new user
with help of Firebase.
Of course, we'll also soon add logic to log users in.
But before we can log users in, we need users
and that's why we'll start with signup
and with creating users.
Now, of course, for that here in the submit method, we have
to find out in which mode we are, if we are in login mode
or in signup mode.
And therefore here, I'll actually adjust this logic
and check if the input was not valid,
in which case I'll just return like this so

that no other code thereafter gets executed so
that I can get this code here out of the if statement.
This is not required,
but make sure that the code stays readable
and I'm not adding too many levels of nested if statements,
which can make the code a bit harder to read
because I'm now going to add another if statement here
after saving this.
I'll check if `isLogin` is true.
So if we are in login mode
and in that case, we'll add the logic to log users in.
Now we can't do that yet because we have no users yet.
Therefore all start with the else branch instead, which is
that we're in signup mode
and here, I wanna use Firebase now to create new users.
Now for that, we can use this Firebase auth package
that we installed.
We can import package and then they are Firebase auth
and from there `firebase_auth.dart`.
Then here in this file, we can add a new global variable,
a final variable here in my case,
which I'll name `_firebase`.
And in that variable, I'll store a value
that will be created with help of the Firebase of clause.
And then there the instance property.
This in the end gives us access to
and Firebase object that will be created
and managed behind the scenes by the Firebase SDK.
And I'm setting this up once here outside of all my clauses
because I can then use that same `fire_base` object over
and over again in my entire code.
And I can for example, use it in the submit method

in this else branch here,
if we are in signup mode to then call various methods
provided by Firebase, for example
the `createUserWithEmailAndPassword` method here, this will do
what the name suggests.

It will create a new Firebase user with email and password.

Now for this method to succeed, you must make sure
that in your firebase project, which you connected
to your flutter project through the FlutterFire CLI,
you got authentication enabled and specifically,
you got the email password authentication enabled
as I did here a couple of lectures ago.

With that enabled,
you can call this method here in your code
and now I wanna pass my `enteredEmail`
and my `enteredPassword` here
to Firebase with that method.

Now `createUserWithEmailAndPassword` will return a future,
which then in the end, will resolve
to a `UserCredential` value.

It may also throw an exception as you learn here if you
for example, already have a user with that email address
or if the email address should be invalid,
or for example if the password should be too weak,
too short, for example.

So therefore because it may throw such an exception and
because it regularly returns a future,
I wanna add `async` here to handle that future thing
and store my `userCredentials` in a variable
called `userCredentials`, after awaiting,
`createUserWithEmailAndPassword`.

But since as mentioned, this could fail,

we also wanna wrap this with try catch
and you'll learn about that early in the course as well.
Here, we can use try catch to try some code
and then catch an exception, an error.
And here, we can now use a slight variation of try catch
and add another extra keyword, the on keyword,
which in the end just allows us
to define the type this error will be of.
And here, we learned that in the end, it will be an error
of type firebase auth exception,
which in the end is just a more specific version
or a specific kind of exception.
And therefore here, we can add FirebaseAuthException
as the type that will be assumed for this error object.
And in this catch block here,
we could then check if error.code is equal to,
and we learn about the available codes here.
So for example, if it's equal to email-already-in-use,
we could then show an appropriate error message
on the screen.
Now here, I'll not do this error.code-specific handling.
Instead, I'll just in general,
use the ScaffoldMessenger here of context
to clear any existing SnackBar
and then show a new snack bar thereafter
with ShowSnackBar just as we did it earlier in the course
to then here, create a SnackBar object where my content
in the end will be a text widget
and the text I wanna show will be my error.message.
Now this might be null and
therefore, here, I'll add this double question mark operator
so that if this should be null,

I show some fallback message,
which could be authentication failed.

So with that, we also have at least some basic
error handling here and we're also getting
those user credentials if authentication succeeds
or if the user creation succeeds to be precise.

And I'll therefore then print those userCredentials here so
that we can take a look at them in the debug console
and see what's inside that userCredentials object.

But that's how we could now use the Firebase SDK
for sending such a user creation request.

Now what's really important to keep in mind is
that you could do this all manually
as you learned it in the HTTP section.

But here, we're using the SDK
to simplify certain things like
for example, this error handling.

The SDK will ensure that such an error will be thrown,
which we can then handle in a quite convenient way
as you see here.

But as mentioned before as well,
of course, we're also using the Firebase SDK
because it will later help us with image upload and so on.

With all that out of the way though, if you saved this all
and you ensure that the submit method is connected
to your elevated button
and you then reload your app, you should be able
to enter an email address here
and also a password and then click on create an account.

That's important because you must be in signup mode
because we right now only added the signup logic
and if you then click on sign up, nothing happens here,

of course, though, that at least proves that we don't seem to have an error. Otherwise, we should be seeing such a SnackBar here. But since we don't see that, it looks like everything worked. And if you now go to the debug console, indeed, in there, you should see such a user credential thing, a quite complex object which contains a bunch of information about the user that was just created. In addition, if you go back to your Firebase project, you should also be able to go to the authentication section and then click on users. And there, you should see that one user that you did just create. And that, of course, proves that the user creation works. Now, if I try to create exactly the same user again with the same email address, I instead get that SnackBar with that convenient error message, which is created by the Firebase auth SDK. And that, of course, therefore, then of course, also shows us that this basic error handling seems to work. And with that, we can now move on to logging users in.

9. Logging Users In

Now before we add the logic for logging users in, I wanna come back to the question how authentication actually works. Of course, you already learned and saw that you collect the user credentials, and you send them to the backend, where they then get stored.

That's what happened in the last lecture,
but now we wanna do more
than just storing those credentials.
Instead now with the logging in,
the collected credentials should not be stored,
but instead validated to make sure
that we have a valid email/password combination,
and Firebase will do that for us, thankfully.
But then what happens thereafter?
Well, in the end,
we might want to change what we display in the app.
We might want to navigate to a different screen
once the user is authenticated.
But in addition to that,
we also might want to be able to access
certain protected resources on the backend,
some data that should only be available to logged in users.
For example, here in this chat app,
the chat messages should only be accessible
by logged in users.
Anonymous users who are not logged in
should not be able to see or create chat messages.
Therefore, once we validated the user data,
the backend also creates such an authentication token.
It actually also does that
when a new user is created, by the way.
Now this authentication token string
is a rather cryptic and random-looking string,
and you'll see one soon,
but in the end,
it will be a string that's generated
by the backend that validated the user,

so that's handling the authentication in the end,
and it's created using a certain algorithm
that actually ensures that the string,
despite looking random, isn't random.
Instead, it is a string that can't be reproduced
by anyone else but that backend
because it uses some secret key,
which only the backend knows
as part of that token-generation process,
but it's then that token
that will be sent back to the frontend
so that it can be stored there.
And that can then be used by the frontend
to attach it to future requests
that are trying to access protected resources.
And since those requests then have
that token attached to them
and only the backend knows how it created that token,
the backend is able to verify that token
and either deny or grant access
to those protected resources.
So that's how the authentication flow generally works.
Thankfully, that token creation on the backend
and that validation of the token on the backend
will be handled by Firebase
since you're not working on the backend,
and the storage of the token on the device
will also be handled by Firebase
since we're using that Firebase SDK here.
If we were not using that,
we would have to store it manually
and manage its lifecycle manually

because it turns out that this token expires
after a certain time period.

With the Firebase SDK,
we don't have to worry about anything like that,
and instead, with that out of the way,
we can now add the login logic
to make sure that users can log in
and that they then also get such a token,
not just when they sign up,
but also when they log in.

And we'll then use that token later
for sending requests to protected resources.

So let's now add the authentication logic.

For this here in this first branch of this if check,
we can again use `_firebase`
to use this Firebase SDK.

And now we don't want to create a new user.

Instead, we wanna log a user in,
and that can be done
by using `signInWithEmailAndPassword`.

Here again, we pass the entered email
and the entered password,
and we learned that this also returns a future,
and that it also may throw an exception
if, for example, an invalid email was provided.

Therefore, here, I'll again await this
and add final `userCredentials` here.

I can use the same variable name as down there
since these are in different branches of the if statement
in the end.

And I'll wrap this with try catch here.

Though, since I want to use try catch here and here,

I can also instead just wrap this overall if statement with try catch

so that I can also reuse this catch logic,

so now the try catch block is outside of this if statement,

or to put it in other words,

the if statement is inside of this try block.

And then we handle errors here,

no matter if they're coming from the sign-up or log-in process.

And then here, when logging in,

I also wanna print my user credentials here to see whether the login process worked.

And that's all we have to do to log users in.

If you now save this and reload,

I can go back here,

enter my email address from before,

and here, I want to use the same one as I did before,

so the valid email and the valid password.

And now here, staying in Login mode,

we can click Login,

and I get no error.

So that's looking good.

And if I now open the debug Console,

somewhere in there,

you should see such a UserCredential output again.

Now you don't see that token in there.

Again, we will see it later.

We don't see it here,

but we can see this output here,

which proves, in combination with the fact that we did not get an error message,

that the login flow also seems to work.

And therefore, with all of that out of the way,
we can now create users and log users in.

10. Showing Different Screens Based On The Authentication State

Now I wanna add more
to this off screen later.

Most importantly, when creating a new account,
users also should be required to upload an image.

So they should be able to take an image and then upload it,
and we'll add this functionality later.

For the moment, I instead wanna make sure
that we show a different screen if we did sign up or log in,
because at the moment we always stay on that screen
and that's of course not the goal.

And to show a different screen,
we can also leverage that authentication token.

Because the existence of such a token in the end proves
that the user provided valid data
because the backend will only create
and return such an authentication token
if the valid credentials were provided.

Hence, this authentication token in the end tells us
in our Flutter app that a user did log in
and I wanna show a different screen if a user did log in.

Now, in addition, I also mentioned that
that token will be stored on the device
and that that storage will be managed by Firebase,
which of course is quite convenient
and therefore it's not just about reacting
to the user logging in or signing up,
but it also means that if we start the app
and the user logged in before when they used the app

before maybe a couple of minutes ago,
we also wanna start on a different screen
and not the off screen again.

So if a user logged in in the past,
we want to keep the user logged in
and not always restart at the AuthScreen.

So ultimately, it's about showing a different screen here
instead of the AuthScreen if we have such a token,
either because a user just finished logging in
or signing up, or because a user is signed up
or logged in in the past.

And therefore here, I'll start by adding a new screen
and that will be my chat screen, the chat.dart file,
which later, as you might guess,
will in the end be responsible for displaying
these chat messages and so on.

But for the moment in there,
I'll just create a very simple class, the ChatScreen class,
which extends StatelessWidget here,
because for now we won't need any state functionalities
in there.

And then here in the build method, I wanna return
a Scaffold widget of course, to get that base screen set up
and that base styling.

And here I want to add an appBar
with the built-in appBar widget where I add a title
that in the end displays a text widget
that could say 'FlutterChat' or anything like that.

And then below the app bar, I'll set up my body.

And here for the moment I'll just use the center widget
as a placeholder to show a basic text here
where I could say logged in.

And of course that will change later,
but that's my content on this chat screen for now.

And now I wanna show this chat screen
whenever the user did log in.

Now to do that, you could think
that here in `authDart` we in the end wait for the user
to log in or sign up,
and we then take a look at the returned credentials
and see whether we got a token in there.

And then we navigate to a different screen,
but we're not going to do any of that.

Instead here I'll just get rid of my print statements
and I'll instead move on to the main dart file
because it's in the end here where
I either wanna show the `AuthScreen`
if we don't have such a token
or the chat screen if we do have one.

And that should then also ensure that if the app is started
and we have a token because the user locked
in before we automatically show the chat screen instead
of starting with the off screen again.

Now to ensure that this all works the way it should work,
I'll replace that `AuthScreen` as my home widget
with an `ever` widget and that `ever` widget will be
the `streamBuilder` widget provided by Flutter.

Now that's a widget we haven't used up to this point,
but it is quite similar to the `future builder` widget
about which you did learn earlier.

You might recall that `FutureBuilder` was used to listen
to a future and then render different widgets based
on the state of their future.

`StreamBuilder` is quite similar,

but the difference between a future
and a stream is that a future in the end will be done once
it resolved, so it'll only ever produce one value
or error, whereas a stream on the other hand,
is capable of producing multiple values over time.
And StreamBuilder wants a stream value, therefore,
and then also a builder function, which will be a function
that automatically receives some context
and then also a data snapshot.
Now just as with FutureBuilder,
this function will be called by Flutter.
Whenever the stream will connect here will emit a new value
and that snapshot in the end will give us access
to that emitted value.
Now the stream I want to connect here is coming
from my Firebase off package.
And there again from the instance
and there we can use the authStateChanges method.
And this method, as you can see here, will return a stream.
So in the end here, we set up a listener
that's being managed by the Firebase SDK under the hood.
And Firebase will emit a new value
whenever anything auth related changes.
So for example, if a token becomes available
or if a token is removed because the user was logged out,
for example.
So auth state changes will give us such a stream
and snapshot, therefore then in the end
will be a user data package, you could say,
created and emitted by Firebase.
So here in the builder,
we can then check if Snapshot has data.

So if we have some user data,
and if that's the case, I wanna return my ChatScreen
because that in the end will mean
that we have a locked in user.
Because Firebase will not emit any data
if we don't have a locked in user,
so if we don't have a token,
Therefore, as an alternative,
if we don't make it into the safe check,
if we don't have data, I will return my AuthScreen.
And here, since I added const here,
I also want to add it here.
But for that on the chat screen, I, of course,
first of all have to add my const Constructor.
This is not a must do, but strongly recommend it like this.
And with that added back in main.dart, we can either return
to chat screen if we have data.
So if we have a locked in user in the end,
that's how this stream created by Firebase here will work.
It will emit snapshots with data if we have a locked in user
and the effort, if we don't have data,
we wanna show the AuthScreen
because then we don't have a locked in user.
With that out of the way, if you save that and reload,
you should in the end end up on this chat screen here
because I did create a user in the past
and I also logged in in the past.
And as mentioned, I got back a token back then
and it was managed and stored by Firebase.
The Firebase estkated all that for us.
Hence that stored token is still there
and therefore here now that we added this logic,

this already stored token will be picked up and leads to the chat screen being displayed instead of the AuthScreen.

And that will even be the case if I quit my emulator here and I restarted, so the same emulator but restarted now and I then run this app on it again.

You will see that as this app starts, we again end up on that chat screen because that token was managed and stored on the device and then automatically loaded by Firebase.

And that's one of the reasons why we're using that SDK, because otherwise we would have to do all these things manually on our own.

11. Adding a Splash Screen (Loading Screen)

So now we're using this Firebase Auth state changes Stream to either show our Chat Screen or the Auth Screen.

Now there's one thing to note.

If I restart this, you might sometimes briefly see the authentication screen before you see the chat screen.

Here, I'm actually not able to reproduce this, but on some devices you might see the auth screen for a fraction of a second.

And this could happen if Firebase takes a bit longer to load that token that is stored on the device.

Therefore, to be safe that you never see this auth screen for a fraction of a second, I do recommend that you add a third screen, which I will add here by just copying the chat screen, which I'll name splash.dart, so that's the file name.

And then the screen in there will be named SplashScreen.

And the idea behind that screen simply is to use it as a loading screen whilst Firebase is figuring out whether we have a token or not.

So therefore, in here I'll output loading as a text, and with that splash screen added back in main.dart in this builder that belongs to the stream builder, you can add a Never if check at the very beginning and take a look at this snapshot that's created by Firebase.

And that snapshot will also have a connection state property just as the snapshots with the future builder had.

This can be used to check whether we're currently still waiting for our first value, so if Firebase is not done loading that token or figuring out whether there is a token yet.

And in that case, we could and should return our splash screen here like this.

Of course, for that, just as for the chat screen, you must add an import here at the top.

So that's what I'm doing. But with all those imports added, we can now show all these different screens for the different circumstances and showing such a splash screen initially until Firebase is done figuring out whether the user is logged in is a practice you might want to pick up and is a feature you might want to add to your apps as well.

With that here, everything works as before, but we're making sure that this auth screen can't accidentally show up initially.

12. Adding User Logout

So now here for this app,
before moving on to image upload,
there's one last authentication related feature
I want to add, and that would be logout,
so allowing users to log out.
Because at the moment, of course,
there's no way for us to get out of this logged in state,
and that's, of course, not ideal.
Therefore, here in this app,
I'd like to add a button here in the app bar
of my chat screen that, in the end, allows users
to leave that app.
And for that here in the chat screen,
so in the chat.dart file, in this AppBar,
we can add the actions list
to add some actions to this AppBar.
And here, it's then just one action I want to add.
It's an icon button that I want to add here,
where the icon will be created with Flutter's icon widget.
And then here it's the `exit_to_app` icon
I want to display, so this icon here,
which looks quite fitting for a logout button.
And I'll actually change the color of that icon
to use a color from my theme
with `Theme.of(context).colorScheme`,
and then they are the primary color, let's say, like this.
And then we just need to add a function here for `onPressed`.
And here, I'll simply add an anonymous function.
And in here I now want to make sure that we,
well, do log out.
And for that we can, again, use `FirebaseAuth` instance.

We could also store that in a global variable,
as I did in the auth.dart file, like this.
Here, I'll just use it like this though,
because that's the only place where I use it here.
And then on that instance, we cannot just call the login
and user creation methods,
but we also have the signOut method.
And that does what the name suggests.
It signs a user out. It locks a user out.
And in the end, that will be all we have to do.
Because since we're then listening to events being emitted
by Firebase here in our main.dart file,
we'll automatically switch back to the AuthScreen
as the user gets logged out,
because Firebase will indeed emit a new event
when the user logs out.
It will emit a new event without data
because no authentication token will exist anymore,
because Firebase will clear that token.
When you call sign out,
it will erase it from the device and from its memory.
And therefore, a new no data event will be emitted,
hence will not make it into either of these if checks,
and therefore the AuthScreen will be shown.
All that will work automatically because of the Firebase SDK
and because of us using its signOut method.
So here if I restart,
and I then press this logout button here,
I am back on the auth screen.
And of course now, if I do log in again,
that also still works.
And with that, we got a finished

basic user authentication flow.

We can create new users,
here using a new email address.

We can then, of course, log out
and log back in with any valid user we created before,
like this.

And of course, we also got the basic error handling,
so that, for example, if I try to log in with a user
that doesn't exist, like this, I get my error message here.

Now you can also see all those users
in your Firebase backend under Authentication, Users here.

You might want to reload this page
to make sure that you see all the users
that have been created.

And you can also delete users from there
if you want to reset your application
and start from scratch again, for example.

But with that, we now get authentication
and we can now move on to image upload,
so that when new users are created,
they also have to upload an image.

13. Image Upload: Setup & First Steps

So now that we added
this user authentication flow,
and we're able to create users and log users in
and also log them out,
I wanna continue working on that signup process.
So on that screen where we create new users,
because the next goal is to add image upload to this app,
so that new users can take a picture of themselves
and upload that picture as part of the signup process,

so that a user is not just a combination of an email address and a password, but instead also has a picture and also a username, which we'll then use later, in combination with the image that was taken in our chat to show pictures and names of users, along with their chat messages.

And therefore, here as a next step,

I want to add image upload to this app.

And I'll again do that with help of Firebase,

and another package provided by the Firebase SDK.

Now Firebase, this backend service we're using here, officially supports uploading files to a Firebase project.

In your Firebase project,

you can go to the storage area here under Build to then later also see your uploaded files.

For the moment, we just need to get started here.

And to get started here, I'll start in production mode, which initially will prevent all uploads, but I'll start in this mode to then tweak those settings to make sure that authenticated users will be able to upload images.

So here, I'll start with that,

and get started with setting up storage on Firebase so that thereafter, we can leverage this feature from inside our mobile app.

And with that storage service unlocked,

we can then go to Rules there to adjust those rules.

And those rules, those security rules here, to be precise, in the end will simply control who's able to upload files to this Firebase project, and who will be able to then view those files.

And here, I wanna make sure that only authenticated users are able to upload and view those files.

And to make sure that that's the case, we should replace false here in this rule with request dot auth unequal null.

This is a Firebase-specific setting that in the end says that reading and writing to any file anywhere in this firebase project should be denied if a user is not authenticated, but should be allowed if a user is authenticated.

Now you can view the Firebase docs to learn more about these security rules and how they work.

If you want to, here, it's this one simple rule that we wanna set up for now to make sure that users can upload and also view those files.

And you also must publish these rules after adjusting them.

And with that, we prepared the storage service, and we can go back to our Flutter app.

And there I now wanna install a new package with Flutter pub add.

And the package we should add here is Firebase underscore storage, which is a package provided by Firebase that in the end is part of the Firebase SDK, you could say, that will be used for sending files to Firebase, which is exactly what we'll have to do soon.

So I'll install this package, and I'll install whenever package here.

And that will be the image underscore picker package, which you already know from an earlier course section, because we used it in an earlier course section

to take images, to take pictures.

And we're going to use it again here to take more pictures

because of course,

the file that should be uploaded to Firebase

should be an image of the user who's using the app.

And for that, we need the image picker library

to make sure that users can take pictures of themselves

from inside our app.

So I'll install this package here as well.

And with that, we now installed all the packages

we should need to add this image upload feature

to this app here,

and that's therefore what we're now going to continue with.

14. Adding a User Image Picker Widget

So, to add this image upload feature,

I'll start by adding a new folder in the lib folder,

and I'll name this folder widgets

because I plan on adding a new widget,

which will not be a screen,

but instead, a widget that's used inside of a screen.

And that will be my user_image_picker widget,

or stored in a user_image_picker.dart file, to be precise.

And in that file, I'll create a new class

which I'll name UserImagePicker,

and we extend a widget class here.

And here, it will be a StatefulWidget that I wanna extend,

because I'll need to manage some state

for picking that image

and for showing a preview of that image before we upload it.

Therefore, here, we of course,

have our constructor, as always,

where I forward that key.

And then we have that related class,

`_UserImagePicker`, which extends `State`,

a state that belongs to the `UserImagePicker` class,

as you learned before.

And in this `UserImagePicker` class,

we add the `createState` method

where we return a new instance of `UserImagePickerState`,

and that should actually be the class name down here,

`UserImagePickerState`.

And with that done here for this state,

the best type annotation

is to be very specific about the fact that,

it's this `UserImagePicker` class

that will belong to this state that's returned here.

And inside of the `State` class,

we then of course, add the `build` method as we always do.

Now, in that `build` method,

I then wanna return a widget that allows me to pick an image

and a widget that shows a preview of the image.

And therefore, here, I'll return that column,

because I'll need two widgets,

which I wanna show above each other.

And the first widget I wanna have in there

should be a `CircleAvatar`,

a widget we did use before in the course,

which in the end, renders a circle

with some content inside of it.

And I'll use this widget for showing a preview

of the taken image.

And below that, I want to have a `TextButton` with an icon

by using this special icon constructor,

which in the end, will trigger the camera to open
to then take a picture.

So here, we have `onPressed`,
and we need an icon.

And here, I'll take the image icon, this one,
and the label will be a Text widget
where I say Add Image.

Now, the icon can and should be `const`.

The Text widget here will not be `const`
because I'll add some style here,
and in that style,
which I'll create with the `TextStyle` class,
I wanna set the color to `Theme.of`
and `theme.of(context)` of course, to my primary color then.

Here in this function that we assign to `onPressed`,
we'll soon open the camera
with the help of the `ImagePicker`.

But before we do that, I wanna work on this `CircleAvatar`.

And there, I'll add a radius of 40,
which in the end,
just controls the shape of this avatar.

And I'll give it a background color of `Colors.grey`,
so of this hard-coded, built-in color.

And I also wanna set
a foreground image thereafter,
and we'll set this dynamically soon
because here, I wanna show a preview of the image
that was taken because this button here was pressed.

So, therefore, as a next step,
we must make sure that this button can be pressed,
and that the `ImagePicker` package
will be used for taking a new image.

15. Using the ImagePicker Package

So, to make sure that we can take images,

I'll add a new method here

to this user image picker state clause.

And I'll name it `_pickImage`.

And in there I want to use this image picker.

I want to create such a image picker object

by using the image picker clause,

which is provided by this `imagepicker.dart` file

from the image picker package.

So, this import here must be added.

On this created image picker object,

we can then call `pickImage`

and define our source.

And here I'll use the image source enum,

which is provided by the image picker package

to use the camera as the source.

Of course, you could also use the gallery,

but here I'll use the camera.

I'll set the image quality to 50

so that we get a small file size

and the max width to 150.

Because here I don't need a big image,

just a small, tiny one,

since I'll not show it as a full screen image

in the app or anything like that.

So, I will only show it as a small picture.

And therefore to make sure

that we don't generate a super large image,

which also takes up a lot of bandwidth

when we upload or download it,

all actually restricted both regarding its size

and its quality so that we get a nice small fitting image.

Now pickImage of course returns a future,

which then eventually yields such as X file.

And therefore here on pickImage, I'll add async,

so that here we can await this operation.

And in the end we'll get the pickedImage here as a result.

And this pickedImage should then be set as a image

that can be used as a preview down there.

So, therefore here I'll actually add a new property

to this class, which I'll name pickedImage file,

and the type should be file.

And for this type to be available,

we must add a new import here from dart io.

And this will be null initially,

hence, I'll add a question mark here to make it clear

that this is not necessarily set.

It may be set, but it doesn't have to be.

And here, once we took an image,

I will of course now set it.

I will set it inside of setState to make sure

that this build method is triggered again.

And in there I'll then set pickedImage file equal

to a file created by using file as a constructor function.

So, this file clause,

which is provided by dart io,

just as I did it in an earlier course section,

and two file, we pass our pickedImage,

and then there the path.

Now pickedImage may be null,

if we closed the camera without picking one.

And therefore here I'll also add a little if check

where I check if pickedImage equals null.

And if that's the case, I'll just return here
and therefore not do anything.
We'll only continue with this step here
if `pickedImage` is not equal to null.
So, then we access its path,
create a file object based on that path
and store that file object in `pickedImage` file.
And with that stored here,
the foreground image on this circle avatar
can and should be set to a `FileImage`
if we have a `pickedImage` available.
`FileImage` in the end yields an image provider
and foreground image wants such an image provider.
That's why we're using `FileImage` here,
instead of the image file constructor function,
which would generate a widget instead of a image provider.
And here we need an image provider.
So, `FileImage` it is.
And to `FileImage` we pass our `pickedImage` file,
but only if it's available.
Therefore here I'll check if `pickedImage` file
is not equal to null,
in which case I'll use this image provider.
Otherwise, I'll set a value of null to foreground image.
And hereafter `pickedImage` file,
we can and should now add an exclamation mark
to tell dart that here,
in this branch of the ternary expression,
`pickedImage` file will not be null.
So, that should display the preview image then.
And therefore now we can save everything,
stop this running process since I did install some packages

that do actually use native device features,
and then rerun the app to make sure
that the entire app bundle is recreated from the ground up.
And therefore once that's finished,
we can then give this a try,
and we should be able to open the camera, take an image,
and also see that image.

Of course, we also must make sure
that this method is connected to the button though.
And therefore, as a last step whilst this is restarting,
all set on press here to pickImage.

So, to this method we just added.
With that, this method here, will be triggered
whenever we tap this button.

And we'll then take this image
and hopefully also show a preview.

Therefore, this widget here is finished
and we can now use this user image picker widget
in the off screen.

And there I want to show it above these text inputs,
but of course, only if we are in signup mode.

Because if we're not in signup mode,
users should not be able to take images.

Because if users are just logging in,
there is no reason to pick an image.

Hence here, in this column
where I have these text form fields,
I'll add this user image picker widget,
but only if we are in login mode.

So, I'll add a if check here using this if statements syntax
that can be used inside of lists.

Where I'll check if not isLogin.

So, if we're not logging in, if we are signing up,
then this will be added.
Otherwise this will be ignored and will not be added here.
And with this added now, with this code added here,
we can now save this, restart the app.
And if we now go to create an account,
we can see this image picker here.
Now I actually wanna adjust the styling
a little bit now that I look at it.
But let's first try whether it works.
And therefore here, if I click on add image,
indeed the camera does open up, I can take a picture,
confirm it, and I see my preview here.
So, that's working.
Therefore, now I just wanna improve the styling a little bit
by making sure that the text color of this button matches
the icon color.
And for that back here on this text button,
I just have to fix a tiny error.
Instead of accessing primary color on the theme,
we should first access the color scheme object
and then primary on that color scheme.
So, not primary color,
but just primary on the color scheme of the theme.
And if we do that, this now looks better.
And that's now this image picker added here.
Therefore, next up, let's make sure
that those pickedImages also do get uploaded.

16. Managing The Selected Image In The Authentication Form

So, to make sure that the pickedImage
is not just shown as a preview here,

but instead also uploaded when we submit this form,
so once we click Signup here,
we first of all, have to make sure that this pickedImage,
which we got through the ImagePicker package,
is communicated to this Form
that in the end stores all these inputs
and all these input values.

And to make that work, here in the auth.dart file,
we should first of all,
go up here where we define all these properties.
And here besides storing my entered email,
entered password and so on,
I'll also add another variable,
which will be of type File,
and which initially at least will be null,
and therefore I add this question mark,
which will be named selectedImage or anything like this.

Now, of course, for this file type to be available,
we again, have to add this import to dart:io.

With that, we got this variable in the AuthScreenState,
and that should now be set
whenever we pick a new image in our UserImagePicker.

Therefore, to communicate from that widget
to the parent widget that contains this widget here,
we'll use the same approach
as we did multiple times in this course already.

We can add a new property to the UserImagePicker,
which will be a function,
a function that returns nothing,
but that it actually should take
one input value, one parameter.

And this argument will be of type File

and I'll name it `pickedImage` here,
and I'll name the property `onPickImage`.
The name, of course, is up to you,
but this property should now be set
whenever this user `ImagePicker` class is substantiated.
Hence, I add a required named parameter here.
Alternatively, you could, of course,
also use a positional parameter,
but here, I'll use a named one, `onPickImage`.
And then with that added here
inside of the `_pickImage` method
in the `UserImagePicker` state,
after updating our preview,
we can reach out to the connected widget
through that special widget property,
and then there, call `onPickImage` to execute that function
that we are receiving as an argument here.
And to that function, we should pass such a file object.
So here, I'll pass my `pickedImage` file
as a value to `onPickImage`,
and add an exclamation mark here
to make it very clear that here, this will not be null,
because we're setting it here in the line
before calling `onPickImage`.
So, therefore, now with this added,
in `auth.dart`,
we can go to the `UserImagePicker` widget here
and add this newly added `onPickImage` argument
to this constructive function call,
and then either point at some method
which we define separately,
or we add an anonymous function here, which is what I'll do.

Where we receive that `pickedImage`,
that's controlled from inside this widget,
and where we then set
`selectedImage` equal to the `pickedImage`.
And that's how we get hold of
that `pickedImage` in this form here.
Now, in the submit method,
I now wanna add one extra validation check.
We are checking all these inputs,
but now we also have to manually check
whether such an image was set.
However, we of course,
only wanna check the existence of such an image
if we are in Signup Mode,
because when we're logging in, no image is required.
So, therefore here, we can add an extra if statement
where we can check whether we're not in Login Mode,
and `selectedImage` is equal null.
So, we're in Signup Mode, but we got no image.
If that's the case,
I also wanna return here.
And we also might want to show an error message
just as we might wanna do it in this check.
Of course, we can also combine all these checks
and add the or operator here
so that here we return
if all these form inputs are not valid,
or if we are in Login Mode
and the selected image is null,
so then we could just use one instead of two if statements.
Either way, I wanna return
and possibly also show an error message if we wanted to.

Here, I'll not do it to save some time.

And therefore, if we make it past this if check,
we know that we have valid inputs,
and in case the user is signing up,
that then also means that we have a selected image.
And therefore, now as a next step,
we, of course, wanna upload that selected image.

17. Uploading Images To Firebase

Now, that upload logic
which should be added here in this else branch
where we create a new user with email and password.
And once we created such a new user,
we then also wanna upload an image for that user.
Now we can't do this in one step
because Firebase simply doesn't support this feature
as part of its auth package.
Instead we first have to create a new user
with Firebase auth package.
And then thereafter we can store any extra data
we wanna store,
like, for example, an uploaded image.
And therefore here, I will start by creating that user.
And then once that finished,
in that else branch here,
we can upload a new image
by using the `FirebaseStorage` object.
And this object here is made available
by the `firebase_storage.dart` file
from the `firebase_storage` package,
which we installed before.
So with this import added,

we can use `FirebaseStorage`
to access this globally available instance
of this class in the end,
which is managed behind the scenes by Firebase.
And then on this instance,
we can call a method called `ref`.
And this simply gives us a reference
of the Firebase Cloud Storage here
that belongs to our project,
so it gives us access to this Firebase Cloud Storage,
you could say.
`Ref` then returns this object
that gives us access
to this storage service in our Firebase project.
And on this object,
we can call `child` to in the end create a new path
in that storage bucket
that is in the end managed by Firebase.
So a new path here in this folder, you could say.
Now, I'll name this path `user_images`,
though this name is up to you.
And I'll then call `child` again on this `user_images` folder
in the end that was created here.
And by the way, if you're targeting a folder
or a file that doesn't exist yet,
it will be created on the fly by Firebase,
so that's quite convenient.
So here with this `child` call,
I'm then diving into this `user_images` folder
that was created in our project
through that storage service.
And here, I don't want to create a sub folder,

but instead a file,
a jpg file here, as it turns out.
And the name of that file will be dynamic.
So I'll dynamically injected into this string
with this special syntax here
about which you learned throughout the course.
And here, I wanna use my userCredentials,
so this object which we're getting back
from calling createUserWithEmailAndPassword in the end.
And on those user credentials on that object,
we'll get a user property
which gives us access to some data
of the user that was created here,
so we can access .user here.
And then on that user object,
we can access uid to get hold of the unique ID
of that user that was created by Firebase.
Now, user could theoretically be null,
but here it shouldn't be null,
it should only be null if this fails.
But in that case,
we make it into this catch block here anyways,
so therefore here we can add an exclamation mark after user
to tell Dart that here it won't be null.
And we'll then therefore create a new jpg file
that carries the user ID of that user
to whom the image belongs as a file name,
which I guess makes a lot of sense.
And with this path created here,
we can then upload the actual image that was taken
to that path under that name.
For that, I'll store this path here in a variable,

a final variable that I'll name storageRef

for storage reference.

And then, here we can use storage ref

and call the putFile method

to upload a file to that path.

Now, putFile wants a file object,

which, of course, is very convenient

because that's exactly what we have here.

This selectedImage here is a file,

and we know that it's not null

because we're having this check here.

So therefore down here for putFile,

we can target the selectedImage

and upload that and add an exclamation mark

because, as mentioned,

here, it won't be null.

And then putFile actually returns an upload task,

which turns out to be a thing that we can await

so that we in the end wait for this upload to finish.

And then we can use that same storageRef

to all the call getDownloadURL,

which then will give us a URL that can be used later

to display that image that was stored on Firebase.

And that's really important.

We're now not storing this image

locally on the user's device,

but instead we're storing it on some remote machine

on a server operated by Firebase

with help of this Firebase Storage service.

And we need this download URL

to then use that file thereafter in our app

so that we can then display the uploaded image.

Now, getDownloadURL returns a future
that eventually yields a string,
and that string will of course be that download URL.
So here I'll await this as well,
and then get my imageUrl as a result.
And for the moment, I'll then just print this URL.
Later, we're going to store it on Firebase
along with some other user data.
For the moment, I'll just upload and print this.
And therefore, now with all these steps taken,
with all these things out of the way,
we should be able to upload an image.
If we now save this and reload
and we create a new account,
we can take a picture here,
see the preview,
and now also enter all these other details
for a user that doesn't exist yet.
So make sure you use an email address
that doesn't exist yet.
And then here,
if I press Signup, that's looking good.
If we now go back to Firebase
to this Storage area
and we reload this page here,
we should see a user_images folder.
And then in there this image here,
which has this user ID as a name.
And if we tap on it, we also see a preview here,
and that proves that uploading the image indeed did work.

18. Showing a Loading Spinner Whilst Uploading

Now, of course,
uploading this image can take a while,
and therefore I wanna show a loading spinner here
whilst we're waiting for this to upload
instead of showing these buttons,
and therefore back here in `auth.dart`,
we can also add another variable, `isUploading`,
which initially is `false`,
but which I'll set to `true`
as soon as we get here into this `try` block.
So here I'll use `setState`
to set `isUploading` to `true`,
or actually, I only want to do this here.
I guess it's therefore not just the uploading process
for which we're waiting,
but also the log in and sign up process,
hence, we might wanna rename this to `isAuthenticating`
here, and also here.
And then, of course, I want to use `isAuthenticating`
to show a different content that we would normally do
if we are authenticating.
Specifically here, I only wanna show this `ElevatedButton`
if not `isAuthenticating`.
So if we're not logging in,
and the same is true for this `TextButton`.
Here, I also only want to show this
if we're not authenticating,
but if we are authenticating,
so if `isAuthenticating` is `true`,
I instead wanna show a circular progress indicator.
So such a loading spinner.

Now one important note about setting
isAuthenticating to true,
so to about managing isAuthenticating state,
we also have to reset it here in this catch block.
Otherwise, if we have an error,
we'll stay in that loading state
and we'll keep on showing that circular progress indicator,
hence making it impossible for the user
to reach those buttons again
and to try different credentials.
So here, after showing this error message,
we should also call setState
and set isAuthenticating back to false,
so that now if we save this and we give this another try,
we can go here, take our image like this,
enter an unused email address,
and now if I click Signup,
I get the loading spinner instead of the buttons
whilst we're waiting for this signup process to finish
and for this image to be uploaded.
And therefore, here,
we then also see this image on Firebase once we're done.
So that's now great, we're uploading images,
but, indeed, that's not the only thing I wanted to do here.
Instead, I also wanted to store this download URL,
which we're getting here,
along with some other data in a database managed by Firebase
so that a user is not just an email and a password,
but also a URL to that image
and a username that can be shown
next to the chat messages later.

19. Adding a Remote Database: Firestore Setup

So, now we're able to upload an image,
and we get this DownloadURL.
The next step now is to store that URL on Firebase,
not here with this storage service,
because it's now not a file that I wanna store.
It's instead, some user metadata,
because of course, I don't just wanna store
Email/Password combinations here,
but also store extra data like that DownloadURL,
and soon also a username that belongs to the user.
Now, at the moment, the only data we're storing
is in the end, the email and password
through that Firebase auth package
when calling create user with email and password.
But this method doesn't allow us
to add any extra parameters,
it just wants an email and a password.
If we wanna store some extra data on Firebase,
we have to do that through an extra Firebase service,
which we also find in our Firebase project under Build.
And there, it's either the Realtime Database,
which we used as a dummy backend before
in an earlier course section,
and we could use it here as well,
but we're actually going to use a different product here.
The alternative, which we could use
instead of the Realtime Database,
and that would be the Firestore Database,
which is a slightly more complex
and more elaborate database solution offered by Firebase.
Now, in my case here,

I'm prompted to go to the Google Cloud Console,
but there, I actually then can choose
SWITCH TO NATIVE MODE.

And with that done, I can then reload this page here,
this Firestore Database page in my Firebase project
to use it from inside Firebase again.

Now, here, we'll then later see
any extra data that we store,
and that's important.

Data, not files.

Instead, data, like the username
or a path to the uploaded image that belongs to a user,
or later, also the chat messages.

Because later, we're also going to use
this Firestore Database for storing the chat messages.

For the moment,

I wanna use it to store some extra user data,
and for that we need to set it up.

And then under Rules,

I wanna make sure that only authenticated users
can read and write to that database,
just as we did it for the storage service.

And indeed here, we should edit that rule in a similar way
to what we did on the storage service.

We should replace false here
with `request.auth != null`.

That then simply means that only authenticated users
can read and write to that database.

And we then should also publish that.

Now, in reality, you might fine-tune these roles a bit more,
because at the moment,
this rule would imply

that every user can read and write any data,
including data that belongs to another user,
and you might wanna prevent that.

So, therefore, you might wanna dive into
the official Firebase Firestore Security Rules
to learn more about how
those rules can be written and structured and improved,
and how you could make sure that users
can only read and write their own data.

Here, for this app, I'll keep it a bit simpler,
also because in this app,
you don't really have a way
of accessing another user's data,
but I wanted to mention
that you could consider locking this down a bit more.

But here for this demo, this basic check will do,
because this check at least ensures
that no unauthenticated users
can interact with that database.

And with that, we got this Cloud Firestore Database set up.

Now to send requests to it from inside our app,
we could, again, use some REST API
provided by Firebase to send HTTP requests manually.

But just as we did it with user creation
and logging users in and uploading an image,
we're not going to do that.

Instead, we're going to leverage another Firebase package
to make our life a bit easier here.

For that, to install that extra package,
we should again run `flutter pub add`,
and then the package name is `cloud_firestore`.

That's the official package provided by Firebase

to interact with the Firestore Database.

And with that package installed,

we can now use it in our project

to send that extra data to Firestore.

20. Sending Data to Firestore

Now, to send extra data to Firestore,

here in the `auth.dart` after uploading the image,

I want to use

`FirebaseFirestore`,

a new object which we can use

after adding this import here to the file,

the `cloud_firestore` import.

And with that added,

we can access `FirebaseFirestore.instance`

just as we used this instance property on `FirebaseStorage`,

and also on this `auth` package provided by Firebase.

And this instance can then be used

for talking to this database here

and for uploading and fetching data.

Now, here it's important to understand

that Firestore works with so-called collections,

which you can think of as folders that contain data.

So on this instance, we can call `collection`

to reach out to a collection

or create it if it doesn't exist yet.

And I'll name this collection here `users`,

because this should be a folder, you could say,

that should contain some user data.

Now, in this Firestore database provided by Firebase,

these collections then contain so-called documents,

which are the actual data entries.

And a document then in turn
could have more nested collections if you needed to,
but here I'll just have
this one root collection called users.
And in that collection,
I'll add a new document by using the doc method,
which can be chained on the result of calling collection.
Here, you then give this document a name,
and this could be a fixed name
or as I wanna do it here, a dynamic name,
because here I want to use that user ID as a name.
So just as I used it here as a file name,
I'll use it here as a collection name
just without the jpg thing at the end.
This will then create a new document
in a collection called users.
And now we just need to tell Firebase
which data should be stored in that document.
And that's done by chaining the set method here
after this document.
And of course, as with all these Firebase features,
you can always dive into the official Firebase documentation
to learn all about these packages
and these Firebase services,
and to learn all about the things you can do with them.
Here, we're using set on this document
to define the data
that should be stored in that document.
And for that here,
Firebase actually wants to get a map
where the keys will be the keys
that will also be used in the database,

and the values will then be the values
that should be stored under those keys in the database.
And here, I wanna store a username,
which is to be done
because that's a feature we have yet to add.
Then also maybe an email,
which, of course, already was stored here
by Firebase when using this auth package.
But we can also store it again here,
the `_enteredEmail` that is,
and then this image URL
stored in a key named `image_url`, for example,
though these key names are all up to you.
And that here, of course, is that `imageUrl` I'm getting here
by calling `getDownloadURL`.
So that's now the data that should be set in this document,
in this collection.
And then here in the end,
`set` yields a future
which we therefore should await
to make sure that we wait
until this operation is completed.
And with all of that added here,
we should now be storing this extra data
in this Firestore database.
Hence, we can save this,
reload the app,
and then log out
to make sure that we can create another new account
with an unused email address.
Also, add an image
(device beeping)

like this,

click Signup.

And now we should actually be able to reload here in this Firestore database part of Firebase.

And I actually don't see the data here because I got an error in my debug console.

Looks like the package actually wasn't found, and that's an error that should be fixable by stopping this process

and restarting this run app process to make sure that all the newly added packages are incorporated into this app bundle correctly.

And here for me, this compilation process actually fails because in the end here for me, it turns out that for Android

I need to set a higher minSdkVersion than 16.

This package, at the moment I'm recording this, needs at least version 19.

So therefore, here to make sure that this works,

I should go back to android

and then there app, build.gradle,

a file we did add it earlier in this course.

And there under android,

in the defaultConfig,

we should set the minSdkVersion to 19.

And besides setting this minSdkVersion,

at least here at the point of time I'm recording this,

I also have to add one other setting

to this defaultConfig in this build.gradle file

in the app folder in the android folder,

and that's the multiDexEnabled setting written like this,

and the casing does matter.

And here, this should be set to true.

With this entry added

and this minSdkVersion set

as mentioned in that error message,

if I now go back,

save everything and run this again,

it should now compile successfully

and the app should run successfully,

and it does.

And therefore, now if I log out,

create another new user here

with email and password

and I add my image as I did before.

If I now click Signup,

this still works, I'm still logged in.

And it might take a short while until it's done,

but then it works.

And now here, if I go back to Firestore database

and I reload this page,

I now do see this user's collection

with one new document that carries the ID

of the user that was created as a name.

And in there, in this document,

I see these different key-value pairs that were stored,

including this image URL of the uploaded image.

And that's how we can store user-specific data,

not with help of this authentication service,

but by simply storing extra data with help of Firestore.

And now, I just wanna complete this data here

by also providing an actual username.

21. Storing a Username

So now in order to make sure
that users also have to provide a username when signing up,
I'll go back to my auth.dart file and there to this form.

And then maybe here
below this email address text form field,
though the exact position does not matter,
I'll add another text form field
and this will be a text form field that's meant to be used
for storing a username.

Now on this text form field here,
I'll therefore add a validator as I did it for the email
and the password as well.

And here we could say that a username should be correct,
if it is maybe at least four characters long.

So here I wanna check if value is equal to null
or if value is empty, or if value.length
or maybe the length after trimming it is smaller than four.

So if either of these conditions is met,
we know that the value is invalid
and will therefore return an error message.

And the error message could be,
"Please enter a valid username, at least four characters,"
or a bit shorter, "Please enter at least four characters."

Otherwise we return null because otherwise,
we have no validation error.

I'll also add a decoration maybe above the validator,
though it doesn't matter.

And here that's an InputDecoration object,
which I'll use to set the label text to a string.

And here, the label simply is username.

And this can be a const object.

I'll set `enableSuggestions` to `false` to make sure that the keyboard on the screen doesn't give the user any suggestions because you typically don't want that when entering fictional names like a username.

Suggestions often aren't helpful there.

That's why I'm turning them off here for this text form field.

And now of course, we also need to add our `onSaved` method or our `onSaved` function where we get the value and where we then save the value just as we did it on the password and the email fields.

Therefore, of course, we need to add a new property to this class here, to this state class.

And that's the `enteredUsername` property which initially holds an empty string like this.

And this should now be updated whenever we save the form, therefore here in this `onSaved` function that belongs to the username text form field,

I'll set `enteredUsername` equal to `value` and add an exclamation mark

because of course here, we know that this won't be null.

That's how we can add such a username field here.

Of course, it should only be shown if we're in signup mode, not if we're logging in.

Therefore here, we should also add a `if` check where we are checking if we're not in login mode.

So if we are signing up and only then this text form field should be added to this list of widget that belongs to this column.

Otherwise, it should not be shown.

If we're logging in, this should not be showing up.

So that's how we then show such a form field.

Of course, this data now also should be submitted once we are submitting the form, so once this submit method is triggered. Therefore here where we're reaching out to Firestore, instead of using to be done as a username, I now want to use the enteredUsername. And now we can save that all, reload, log out again. And we don't see the username field here when logging in, but if I create an account, I now do see it here. So now again, we can take a picture here, use an unused email address and a password. And then here also a username like max1. And if I now sign up, this takes a short while, we're logged in and now in Firestore, we see a new document and there, we now got this username stored as well. So that's this feature added.

22. Adding ChatMessages & Input Widgets

So now that we've got the entire authentication process finished, and now that we even got started with Firebase Firestore with that extra database we're also going to use for storing the chat messages, it's time to start working on that chat. And for that here, when logged in, I, of course, wanna make sure that on this screen we show the chat messages, but also a little row at the bottom that has a text input field and a button that allows users to enter a new chat message.

So for that, here on this chat screen,
this dummy Center widget should be replaced
with a Messages widget that shows all messages
and a NewMessage widget
that allows users to enter a new message.

For that, here in the widgets folder,
I'll add a new file, chat_messages.dart,
that will later show that list of chat messages,
and also a new_message.dart file
that will show the form for providing a single new message.

Now, the chat_messages.dart file
should contain a new class that can be called ChatMessages.
That should extend a StatelessWidget
because here we won't need to manage
any state in that list of messages.

We'll just load that list of messages,
but we'll then display that list
without manually changing any state,
and you'll see how that works a little bit later.

Therefore, here we're extending StatelessWidget.
We can and should add our constructor here
where we get and forward that key
as we did so often in this course,
and we should, of course, add that build method.

And then here in this build method, at the moment,
I'll just return a Center widget,
which has a single child as all Center widgets do,
where I output some text and I say, "No messages found."
And we'll of course replace this later.

For the moment, that's the content
displayed by ChatMessages.

Next, I also wanna work on the new_message.dart file.

And in that file here, we add another new class, `NewMessage`, and this will extend `Stateful widget` because here we will need to manage some state, and we, of course, have our constructor here, as always, `super.key`.

And then we have our `createState` method, and we have that state class here, underscore `NewMessageState`, which extends `State`, a state that belongs to the `NewMessage` widget. That's also the state we wanna set up here.

And then here we return a new instance of this `NewMessageState` class here like this.

And here in the `NewMessageState` class, we then add the `build` method.

And in this `build` method here, you can then, of course, style this however you want. You can use whichever widget combination you want. I'll simply add a `Padding` widget here to make sure that we do set up some padding around the text input field and the button.

And here I'll set my padding to `const EdgeInsets.only` to set specific padding values for the different directions. And use a padding to the left of 15, to the right of 1.

And you'll see soon that this should look decent. Of course, you can tweak those values as needed, though, and set a bottom widget of 14.

But again, you can definitely fine tune and tweak those settings.

Then here as a child for `Padding`, I wanna use a `row` because I wanna have two widgets next to each other horizontally.

And the first widget should be a TextField,
not a text form field,
because here I'm not using the form widget,
though, of course, you could also use that if you wanted to.
And TextField inside of a row can be problematic
as you learned before in the course.
Hence, I'll wrap it with expanded like this.
Now, this Expanded TextField combination
should then go into the children list of row,
but it should not go in there alone.
Instead, in addition here, I will also add
an IconButton widget to render,
well, an icon button
where the icon will be the send icon,
so the send icon it is.
And then onPressed here is, of course as always, a function
that will be triggered whenever this button is pressed.
Now here, I'll then also add a color,
and that will be Theme.of(context).colorScheme.primary.
And now we just have to configure this text fill here.
And there, for example, I wanna set textCapitalization
to TextCapitalization.sentences
to make sure how the device supports the user
with entering text.
That it, for example, should capitalize the start
of every new sentence, which you can turn off
or configure differently,
but it is a configuration I wanna add here.
I'll also set autocorrect to true to make sure
that errors are caught and corrected automatically.
I'll enable suggestions by setting this to true like this.
And I will also add a decoration here,

an InputDecoration, of course, to set the labelText
to send a message, dot, dot, dot.

And this can be a const object here, of course.

Now last but not least, on this TextField,

I'll also add a controller

to get hold of the value entered by the user.

And for that, of course, here in NewMessageState,

I'll add a new messageController,

which is a new TextEditingController object
created like this.

And as you learned before in the course,

you should also add the dispose method

when dealing with controllers.

And then here you should call messageController.dispose

inside of this dispose method

to make sure that the memory resources

occupied by the controller get freed up

as it's not needed anymore.

And it's this messageController that's used as a value
for controller on this text field.

And now whenever this button is pressed,

I wanna read that entered value and send it to Firebase.

Therefore, here I'll add a new method,

submitMessage could be the name.

And this method is now connected to this button

like this so that the submitMessage method

is triggered whenever this IconButton is pressed.

And in submitMessage, we can then get the enteredMessage

by using the messageController

and accessing the text field,

which will return the entered text as a string.

Now that we get hold of the entered text here,

we can add validation
and check if the enteredMessage,
once we trimmed it, maybe is empty,
because empty messages should not be sent to Firestore.
So in that case, I'll return here
and we also might want to show an error message,
otherwise it should be sent to Firebase
and we'll add the logic for this soon.
And thereafter, I wanna use my messageController again
to call the clear method and reset that input
so that this text field is set back
to an empty text field again,
because, of course, we're not going to leave this screen
or anything like that.
Instead, this NewMessage widget will soon be rendered
below the ChatMessages widget.
So it will always be visible,
that's why I just wanna reset it and clear it
after a message has been submitted.
And with all that added here, back in chat.dart,
so in this chat screen widget,
here, instead of displaying this center widget,
I now wanna output a column
where I have two child widgets
and the first widget will be my messages,
my ChatMessages widget.
So this widget here in the end,
which at the moment is this Center widget,
though that's going to change in the future.
And then below that, I wanna have my NewMessage widget.
Now, both can be created with const,
and if that's the case, you can also add const

in front of the list, so in front of the square brackets.

With all of that out of the way, if we save this and reload the app, we got no messages found here, and then our input field and this button here.

Now I'm not 100% happy with that yet.

Instead, I'll wrap ChatMessages with Expanded here, not to get rid of some error,

but instead to make sure

that this ChatMessages widget indeed does take up as much space as it can get

and we push this input here to the bottom.

But with that, that's looking good.

And now here I can enter something, submit this, and thereafter, this is cleared just as I want it to work.

So that's looking good.

This seems to work the way it should work.

Now the next step is to make sure that any message that was entered is indeed also sent to Firebase.

23. Sending & Reading Data To & From Firestore

So, how can we now send data to Firebase

from inside this new message.dart file?

So, from inside this _NewMessageState widget here.

Well, we're extracting data from that messageController,

and therefore, we can actually, also,

set this to a final controller here

to get rid of these blue squiggly lines.

But we are extracting text from it,

we're validating it here to send it to Firebase.

We can then, again, use FirebaseFirestore.

So, this object here

provided by the cloud_firestore package,

which we installed before, and which we also used before
in `auth.dart` for storing that extra data.

And we can now use exactly the same approach
for storing a new message.

So, we, in the end, want to get hold of the instance here,
then reach out to a collection,
and to some document to store new data.

So here, in this new message state class,
we reach out to the instance of `FirebaseFirestore`,
then to some collection.

And here, I'll name the collection `chat`,
though the name is up to you.

And then, here, we could create a new document like this.

But here, I'll instead use a different approach.

I'll call the `add` method to dynamically add a new document,
which then does not have a name of my choice,
but which instead will have a unique ID as a name,
generated by Firebase.

This `add` method then takes another map,
and that map then again contains these key-value pairs
that should be stored in Firebase.

So, the only difference
compared to the approach taken in the `auth.dart` file
is that we don't set our own name
explicitly for this document,
but that we instead go with an automatically generated name.

This automatically generated unique ID
that will be created by Firebase.

Now, the data that should be stored in this document,
of course, should be the entered text,
but also some additional metadata.

But I'll start with the text, store it under a `text` key,

though that key name is up to you,
and it's the `enteredMessage` that should be stored here.
But in addition, I also want to add a `createdAt` field
to store the current timestamp.
And for that, we can use the `Timestamp` clause,
which is built into Dart,
and then use `now` to use the current timestamp.
And in addition, I wanna store some user data,
like the ID of the user to whom this message belongs.
So, who posted this message.
And to get information about the currently logged in user,
we can use `FirebaseAuth` again.
This `FirebaseAuth` thing,
which we set up before in `auth.dart`.
So, we again want to use `FirebaseAuth`,
this `FirebaseAuth` object, which is available
as soon as we add this `FirebaseAuth` import here.
And then again, we can use `.instance`
to get hold of the globally managed instance.
And the convenient thing about `FirebaseAuth`
is that it does not just give us these methods
for creating users or for logging users in,
instead, it also gives us access
to the currently logged in user if we have one.
And of course, here we know that we will have one
on this chat screen,
because we only reach this chat screen if we are logged in.
So, this gives me access to the currently logged in user.
And then here on `userId`, I can use `user.uid`
to get the unique ID of the currently logged in user.
Now here I just need to convince Dart that `currentUser`
will not be null, by adding an exclamation mark.

And again, as mentioned here,
we know that this shouldn't be null,
because unauthenticated users
shouldn't be able to even reach this screen.
So with that, we can get the `userId`,
but then here, I actually also wanna store the username
so that we can easily show that next to the message,
when we're outputting the messages later.
And I also wanna store the user image,
so that we also have that image URL
readily available when we need it.
Now that data, username and user image,
of course wasn't stored with help of `FirebaseAuth`.
Instead we stored it manually with the `Firestore` package.
Therefore, now to access it here,
we need to retrieve it from `Firestore`.
And this can also be done
through `FirebaseFirestore` instance.
And then there, we again reach out to a collection,
the users collection in this case,
because that's the collection we also used
for storing the data.
And then we wanna reach out to the document that belongs
to the user with this id.
So therefore, again, we can use `doc`
and then `user.uid` to get hold of that document,
which we know contains the data we're interested in.
And then `Firestore` gives us a `get` method,
which then in the end will send an HTTP request
behind the scenes to `Firestore` to retrieve the data
stored in this document in this collection,
which in this case will be that user data

that we stored there.

Now get in the end returns a future,
which then in the end will give us access
to the fetched data.

Therefore here, to use async await,
I'll add async here on this submitMessage method.

And then we can await this here.

So, here we get the userData, by awaiting,
this get request here in the end.,
and then here we can set username equal to userData.
And then, this actually is a so-called document snapshot,
which is a type of data provided by Firebase.

And this object has a data method which we can call,
to in the end, get access to the actual data
that was stored in the document as a map.

We have to add an exclamation mark
to convince Dart that this will not be null.

It could be null if we're trying to fetch data
for a document that doesn't exist, for example.

But here we know that we have an authenticated user,
and therefore it will exist.

So, we can add the exclamation mark here,
and then we'll be working with that map,
so we can use square up brackets,
as you learned in this course,
to access the different values for the different keys
that make up the map.

And then here it's the username key which I want to access
to get the username value.

And I'm using username as a key
because I used username as a key for storing that data.
And I used image_url for storing the image URL data.

Therefore, for retrieving that, we would use `userData.data` with an exclamation mark and access `image_url` like this.

And that will retrieve this extra data from Firebase Firestore.

Now, we could also be storing that extra user data locally on the device, maybe just in memory, for example, with Riverpod.

But here, I'm deliberately using this remote solution using Firestore, even though it leads to this extra HTTP request being sent whenever we submit a new message.

But I'm using this so that you also see how data can be retrieved from Firestore, because that's of course also something you should know.

With that, we're getting the data, we're then using it to send it back to Firestore combined with the entered message.

And therefore, now with all that out of the way, if we saved it, reload and make sure we're logged in with a user where we did submit that extra data to Firestore.

So, if that's all true, we can now enter a new message, like a test message, submit it, and now here, we should actually see a new collection if we reload this, a chat collection with one new document.

And this title was generated automatically by Firestore, by Firebase in the end.

And in there, we see that submitted data:

the username, the userImage, the ID, but of course, most importantly, also our message.

So, that's all working as it should.

Now the last thing I wanna do, is I wanna make sure

that once I send a message, I don't just reset the input,
but I also close this keyboard.

And this can be done by going here in my code
after clearing my message controller.

And then here we can use the FocusScope object
provided by Flutter, of(context),
and then call .unfocus.

And in the end, this will close any open keyboard
by removing the focus from the input field.

Now here, I'm getting some blue squiggly lines
because I'm using context after async await.

We could work around that as we did early in the course,
or we simply move that

to the beginning of this async method,
because indeed I am happy with closing this right away,
maybe here after validating the input.

Because we don't need to keep this open
whilst the message is being sent,
instead, closing it immediately is just fine.

Indeed, we can even clear the input there already,
because even though the data was not sent yet,
we definitely don't want the user
to send the same message twice
because they think that something went wrong.

So, clearing this immediately,
and losing the focus immediately, makes a lot of sense
before then the data is sent behind the scenes.

So now with that, if we reload this,
I can send another message.

And now you see this is closed, this is cleared.

That looks good.

And of course, these messages all show up here

on the backend.

And that's how data can be sent to Firestore

and how it can be read from Firestore.

And now that we know how to read it,

we can of course next work on the chat messages,

and make sure that we load all the chat messages

to then show them in a list of messages in this widget.

24. Loading & Displaying Chat Messages as a Stream

So let's now fetch

and display all these stored ChatMessages

from inside this ChatMessages widget here.

For that here, in this build method,

I wanna return a StreamBuilder widget.

And you learned about the StreamBuilder before,

you saw it in the main Dart file.

There, we used StreamBuilder to be notified

about any auth changes.

So a user logging in or logging out, for example.

Now I want to use it to listen to a stream of messages

so that whenever a new message is submitted,

it's automatically loaded and displayed here.

And we can use StreamBuilder here because Firebase

and these Firebase packages support this stream concept.

For that, we need to set the stream parameter here

of the StreamBuilder to a stream value

and that stream value will be provided

by FirebaseFirestore.instance.

And then there we can reach out to a collection,

the chat collection,

because that's the name I chose

for storing all these messages.

And on that collection we can call snapshots
and snapshots will yield such a stream.
Therefore we can use it as a value for stream here.
And in the end, what this will do is it'll set up a listener
which automatically listens to this remote database
and there to the chat collection.
And whenever a new document is added here,
it'll automatically notify our app here
or this Firebase package will automatically be made aware
of that to be precise.
And it will then trigger this builder function here again
so that we can update the UI
whenever a new document is added to this collection.
Therefore here of course we must add a builder method
and this builder method
will get a context object automatically.
And our chatSnapshots object.
Both values will be passed in automatically
by Flutter in the end by Flutter working together
with the Firebase package, you could say,
and chatSnapshots will be in the end an object
that gives us access to the data
that was loaded from the backend.
So from Firebase.
Here in the builder,
we can then use this chatSnapshots object to, for example,
evaluate the ConnectionState.
And if that's equal to ConnectionState.waiting,
we know that we're still waiting to receive some data.
In that case, we could return center widget
that displays a circular progress indicator, for example,
so that we show a loading spinner

whilst we're waiting for ChatMessages to arrive.

We also might end up in a state

where we have no ChatMessages.

So here I want to check if chatSnapshots hasData is false

by adding an exclamation mark at the beginning

or if chatSnapshots data, if we have data,

if that is actually an empty list

if we have no documents in our collection.

For that, we can use the docs property here

and docs will in the end yield a list

and we can check if that is empty.

Now, data here theoretically could be null

though it won't be if hasData is true.

So if this condition is checked,

we know that data won't be null.

So then I can add an exclamation mark here.

But of course the docs could be an empty list.

So therefore here I'm checking

whether we maybe don't have data despite being done

with loading or we have data, but it's an empty list.

And if that's the case, I wanna show this here

as a fallback message.

Ultimately, if we make it past both if checks,

we know that we have data,

and then I wanna return a ListView

using the ListView.builder constructor

to build such a performance optimized

scrollable list of data.

By the way, we also might want to check for errors

by taking a look at chatSnapshots hasError.

And if that's the case, we might to output the error message

or at least output a text widget that's centered

where we say something went wrong.

But thereafter we can display our ListView.

And then here we have this builder function

where we also get this context object

and then the index of the item

that is currently being rendered.

And then here we have to return our list items.

We also have to set a itemCount.

And that itemCount

of course is based on our documents here.

So here we can use chatSnapshots.data!.docs

to get our loadedMessages in the end like this.

And we know that data won't be null here

because otherwise we would've made it into this

if check here.

And then it's loadedMessages.length,

that's our item count here.

And then here for a start to keep things simple for now,

for every item in this itemBuilder function,

I just wanna output a text widget

where I simply output the text that was stored

for this chat message.

So that was stored in this text field here.

And that can be done by accessing LoadedMessages

for the index for which we're currently rendering an item.

And then there we can call the data method

to get access to the underlying map

and therefore on that map

we can then use the square brackets

to access the text field.

So the value that's stored under a key text in that map.

If we save that and we reload,

we should see these ChatMessages here
on this screen.

I also want to make sure that they are always ordered
like that the order is actually correct here,
but to make sure that it will always be correct.

We can also fine tune this query here,
which we assigned to stream.

And before calling snapshots after a collection,
we can squeeze in the orderBy method on which result
you can still call snapshots to define
how these collection documents should be ordered.

An order by allows you to specify a key
that must be present in your document.

For example, createdAt, which is what I'll use here.

And then as a second parameter value,
as a second argument after createdAt,
you define whether you wanna sort an ascending
or descending order.

And here I'll set descending to false
to in the end make sure
that the latest message will be displayed at the bottom.

With that, I got the same order as before,
but now I'm making sure that I will always have this order.

And most importantly, you learned
about the orderBy method here.

But that's how we can now load
and display that data with help of FirebaseFirestore
and the StreamBuilder.

And whenever we add a new message like this should work,
you see that's added here automatically.

We don't have to manage any state manually,
we don't have to do anything like that.

We just submit the data.

And then thanks to this stream here, so to this listener,
which we in the end have year,
it's all updated automatically thereafter.

25. Styling Chat Message Bubbles

So now that we're able to load
and display those chat messages here,
it's time to make sure that they're styled properly
and that we, for example, also display the user's image
and username next to them.

And I want to start with some general list styling here.

On this list view here, it, for example,
would look a bit nicer if we would add some padding.

And thankfully, the `ListViewBuilder`
constructor function supports a padding parameter.

For that here,

we can then create such an `EdgeInsets` object using
the only constructor function again
to set specific paddings for the different directions.

And here I want to set a bottom padding of 40
to have quite some spacing between the list of messages
and this new message input area.

And to the left and right, I want to have a padding of 13.

And of course, as always,
definitely feel free to tweak those values to your liking.

Now, one other thing which is quite important,
which I want to do here,
is I want to set the `reverse` parameter to `true`.

Now, what does this parameter do?

If we save it,

you can see that now the messages are pushed to the bottom.

In the end, the entire list is reversed.

And instead of going from top to bottom,
it now goes from bottom to top.

Now, of course, I still want to make sure
that my latest message,
which in this case would be this one here,
is at the bottom of that list, though.

And to make sure that that's the case,
we can use this `orderBy` method again
to order these chat messages in a different way.

Instead of ordering them in ascending order,
we can order them in descending order
by setting this to `true`.

And combined with that reversed list,
which I did do to make sure that it starts at the bottom,
we will now have the correct output again if we save this.

And we have the latest message at the bottom,
but now the overall list is pushed down to the bottom.

Now, of course,
these list messages also should look a bit nicer.

Instead of just displaying a text widget for every message,
I want to, in the end,
display a chat bubble that contains
the message combined with some extra metadata next to it.

And for that attached,
you find the `message_bubble.dart` file,
this file here, which contains all the widgets
and all the styling you need to display such a chat bubble.

Now, it's a lot of code in there,
but it does not contain any features
we wouldn't have covered before in this course.

And therefore, to not waste your time,

we're not going to build it from scratch together.

Instead, I'm just providing this finished widget to you and you find a lot of comments in that widget that explain the different main parts and main settings that make up this overall custom widget.

In the end here, we're just rendering a chat bubble, which contains the chat message.

And depending on whether it's the first message of a user in a sequence of messages or not, it will also output the user's image and also the username next to the chat message.

Now for that, this message bubble offers two constructor functions, `first` and `next`, which should be called depending on whether it's the first message in a sequence of messages by the same user or some other message, not the first message of the same user.

And then we must provide the user image, username, message and whether that user is the logged in user because the messages of the logged in user should also be displayed in a slightly different way.

So with that widget added here in `chat_messages.dart`, we can use this widget in our builder method instead of displaying such a text widget, because instead of doing that in this builder method here, we can now get hold of that chat message by using the loaded messages for a given index, as you learned, and then they are calling `data` and that gives us this document map, which in the end contains all that metadata that belongs to a message and the message itself.

And now for every message that will be output,

I also want to take a look at the next message in line,
if there is any, to see if that next message is
by the same user or not,
because as mentioned a couple of seconds ago,
that will be some important information
for displaying this message bubble widget correctly.

So therefore, we also want to have a look
at the next chat message and we can get that look
by using this index here of the current chat message
and adding one to it, which will then be the index
of the next chat message.

And we want to check if index plus one
is still smaller than loaded messages dot length,
because if it's not smaller,
it means we're trying to look at a message
that doesn't exist.

We're trying to exceed the boundaries
of our list of loaded messages.

But if it is smaller,
we know that there is a next chat message in line,
that this is not the last chat message here.

And in that case here, if there is a next message,
we will get hold of it by using loaded messages
for index plus one.

Alternatively, we'll yield null.

We'll return null.

So next chat message is either null
if there is no next chat message
or it is that next chat message in line.

Now with that next chat message loaded here as well,
we can then get the current message username
by using our chat message.

So, this chat message here,
which is loaded for the currently displayed index.
And there we can access the username key
because in the end for every chat message,
we are storing the username.
But we can also take a look at the next message username
to see if that next message will be from the same user.
And for that, we can use the next chat message.
And if that's not equal to null,
we can access next chat message username.
Alternatively, we return null.
If there is no next chat message,
then we set next message username to null.
And with that, we could then compare these usernames.
Now, theoretically,
two users could have the same usernames though.
Therefore, we might instead wanna use the user IDs.
So, the current message user ID
and the next message user ID.
And conveniently, I'm also storing those user IDs along
with the text that makes up a message.
And therefore, we can retrieve those user IDs here.
By using that user ID key here.
And that then will give us a unique identification criteria.
And we can then add one last utility variable here
that could be named next user is same.
And that should be true if the next message user ID
is equal to the current message user ID.
So, with that, we're simply checking whether
the next message is from the same user
as the currently rendered message here.
And that will only be important for outputting

the message bubble correctly to get the styling of that bubble right by using the write constructor function of this message bubble widget.

Because now we can add a if check and check if the next user is the same.

And if that's the case,

so if the user is the same as the current user,

we want to return message bubble next

because we have an ongoing sequence of messages from the same user.

In that case,

we want to output chat message text here

as a message text.

And we need to find out whether this user

for whom we're outputting this message

is the currently logged in user.

And to find out whether that's the case,

we can go back to the beginning of this build method maybe

because we don't need to get information

about the currently logged in user for every message

that we're outputting.

Doing this once when we're starting to output

the overall list is enough.

And here we can get the authenticated user

by using firebase auth instance current user,

as you learned.

And current user could be null,

but here it won't be because again,

we're in a part of the app where only logged in users

have access to.

So that's our authenticated user.

And down here where we're outputting the chat messages,

we can therefore then use this authenticated user
to get this user's ID
and compare this to the current message user ID.
And if that's equal, we know that the currently logged
in user is the user who created this message.
And then is me should be set to true
to change some style settings in this message bubble widget.
Again, you can dive into the message bubble widget
and take a look at all these widgets to understand
which settings are changed based on which parameters.
For example, where this is me, parameter is being used.
Now, of course, we also have to cover the case
that the next user is not the same.
So, the else case here.
And in that case,
I wanna return message bubble first
because it's the first time
that we're outputting a message for the given user.
And in that case, we of course also still provide
the text and we still provide the information
about whether this is the logged in user or not.
But in addition, we also need to provide a user image
and the username.
And that can be done by using chat message
and then their username.
And of course,
also chat message and then the image
is stored under user image in my case here.
So that's the key,
this key here that I wanna use for extracting
that image like this.
Now with that,

I just noticed that I mixed this up.

So of course, that's the user image, that's the username.

But with that, if I save this all and reload,

I now got the username here above my messages.

And then next to my messages,

I also got the image that belongs to this user.

And now if I do log in with another user

or create a new user here, which I'll name Maximilian.

So that's a different username than what I had before.

And I add a different image here,

maybe of that TV here on this emulator.

And I sign up.

I now see these messages here to the left

because now they belong to a different user.

But if I enter my message here,

it shows up on the right with my image.

And if I enter more messages here,

they show up in the same sequence.

So now with all of that,

we have this app where users can log in,

can view messages and also send messages

and where we then also display those messages

on this screen.

Of course, definitely feel free to tweak the look

of this app if you want to.

For this section here,

the last big feature

that should be added are push notifications though.

26. Push Notifications - Setup & First Steps

So now that we worked on this chat screen

and we are able to output these chat messages,

there's, in the end, only one feature

that's missing in this app.

And that would be push notifications.

I wanna make sure that users receive a push notification

whenever a new message arrives.

So if the app is not opened

and a new message is written by some other user,

a push notification should arrive.

And when users tap that notification,

the app should open up.

That's the goal.

And thankfully, that's also something

that can easily be implemented with help of Firebase.

There, in your Firebase project, you have this Engage area

and there, you have this messaging feature,

this messaging service.

And that's exactly the service we're now going to use

because this messaging service can be used

to send push notifications to both Android

and iOS devices.

Now to get started with Firebase push notifications

and Flutter, it's best if you search

for Flutter Firebase push notification or push notifications

because that will lead you

to the official Firebase documentation

and there, to an article, which I also attached

to this lecture though, which in the end, walks you

through all the required setup steps you should go through

to make sure that your app is capable

of receiving push notifications,

because especially for iOS, there are a couple

of steps you must go through to ensure that you can receive

and handle push notifications.

So therefore now, we're going to walk through these iOS and Android setup steps,

though these iOS steps will only matter

to you if you're on a Mac OS device.

Otherwise, you won't be able to build for iOS anyways

and you won't be able to configure your project

for receiving push notifications.

It is unfortunately as it is,

that is a restriction applied by Apple.

So to set up push notifications for iOS devices,

you should open Xcode, which is an app

provided by Apple for Mac OS.

And if you don't have Xcode on your Mac OS device yet,

you can install it

through the App Store on your Mac OS device.

And with Xcode opened,

you then should open an existing project or file

and you should navigate to your chat app folder,

to your Flutter project folder, there to the iOS folder.

And then as mentioned here in the official Firebase

documentation, open the Runner XC workspace file

in that iOS folder.

So this file here, open it with Xcode,

and once you did open it, you have

to enable push notifications,

as described here on this external link.

Now for that, you need to do a couple of things.

For a start, you should select your Runner project here

in Xcode and under Signing and Capabilities,

you should click on Plus Capability.

And then there, search for push

and add push notifications by double clicking on it.

Now once you do that, you will most likely get an error here

and you might've even gotten an error here in

that signing area before even adding push notifications

because you now must assign a development account,

a development team to this iOS app

that you're building here.

And for that, you actually need

to create an Apple Developer account.

You also learn about this here in the official

documentation.

And here, you also find more information

about this Developer account and a link

to that account page.

There, you can sign in with your Apple ID.

And once you did that,

you can create your Apple Developer account.

Now for certain features like

for example, publishing your app in the Apple App Store,

you will actually need a paid Developer account.

So you will need to pay a yearly fee

to use these extra features.

For testing push notifications

with a local device, with your own device,

you won't need that though.

And that's, by the way,

all important push notifications on iOS can only be tested

with a real device, not with a simulator,

but I'll get back to this later.

So you need to create such a Apple Developer account,

but you won't need to pay for it.

And with such an account created,

the main error you might have gotten before here, where you might have not had a team selected here, should go away.

Now you might still get another error here that for example, the bundle identifier is not available, which is the case for me here.

To fix this, you can tweak this identifier here and for example, rename it to chat app letter test or anything like this.

And then try again until Xcode succeeds setting up this project for push notifications.

That's the first step then.

We will also need this Developer account again later, but that was the first step.

As a next step, as mentioned in the official documentation, you also must enable Background Fetch and Remote Notifications in your project.

This is done by adding another capability and that's the Background Modes capability, where you then should select Remote Notifications and Background Fetch as mentioned.

Once you did that, there's one last step left.

You need to go to your Apple Developer account and there under Certificates, Identifiers and Profiles, you should go to Keys.

In that Keys area, you then should add a new key and you should choose Apple Push Notifications Service.

Now in my case here, I have too many keys.

So first of all, get rid of some of the existing keys, which I don't need anymore.

And with that done, I can create a new key and now select this Apple Push Notifications

Service Key here.

This will be required for making sure
that push notifications can be sent to iOS devices.

Now you must give this key a name like Flutter chat app test
or anything like this, and then click Continue and Register.

And then you should download this key.

So it's in the end, a file that gets downloaded here.

Now that this key was created
and downloaded, we can move on with the next steps here,
where we have to upload this Apple push notification
authentication key to Firebase.

For that, as we're instructed here,
back in our Firebase project console, we should go
to project Settings
and then there to the Cloud Messaging tab.

So here in Firebase, we can click on this gear icon
to go to Project Settings.

And then there, Cloud Messaging.

In that Cloud Messaging area, we should select
Upload Certificate.

So in the end here in the Project Settings, we should go
to this iOS area and then select Upload here.

And then look for that key file you just downloaded.

So in my case, this file here.

Open it and then enter the key ID,
which you do find here in your Apple Developer account.

And add this key ID here and then enter your team ID.

And that can be found in your Developer account as well.

You find it in the top right corner.

Of course, you should use your team ID here
and this should now be entered here as well.

And then you can upload this key.

And with that APN authentication key uploaded here,
we're finally done with that setup step here.
Please note that here in the end, it recommended
to upload a certificate,
but as you saw in the Firebase project page,
uploading such an authentication key was recommended.
And that's the approach I chose here.
So therefore, we're now done with that, finally,
and we just have to finish the Android setup.
And there, thankfully it's much easier
because there for one,
we can also test push notifications on an emulator,
not just on a real device, as it's the case for iOS.
And we also don't have to go through any extra setup steps.
So that's quite straightforward.
And therefore, the only remaining step here is
to install the Firebase Cloud Messaging Plugin
by running this command in our Flutter project.
So here in the terminal,
I'll run `Flutter Pub add Firebase Messaging`.
And with that done, I will shut down the running app process
and restart it from scratch to make sure
that everything works correctly there.
And we should therefore now be ready
to also add some code to our application that allows us
to receive push notifications.

27. Requesting Permissions & Getting an Address Token

So let's now add
some code to receive push notifications.
And for that,
I'm back here in this chat screen widget

because in the end it's this widget here
where I want to set up this
push notifications logic,
and it's this widget instead of the auth screen
or the app widget here, for example,
because I know that users were locked in
once they are on the chat screen,
only authenticated users reach this screen,
and push notifications
only matter to authenticated users
because in this app here
I am sending push notifications
whenever a new message was created.
And since those messages can only be sent,
and viewed by authenticated users,
it doesn't make much sense
to set up push notifications,
and ask the user for permission
to receive push notifications
before they were locked in at least once.
The efforts in this chat screen
where I wanna set up push notifications,
and set up push notifications
in the end means that
we want to ask the user for permission.
That's especially important for iOS devices.
And we also need to get the address
of the device on which the app is running
because that address will be important
for sending push notifications thereafter.
Now to set up push notifications here
in the chat screen,

we should start by converting
this stateless widget
to a state full widget
because I need to use this
in its state method here, again,
which we can use in this state clause.
Because this method will only run once
when this widget is first loaded in the end,
and it's there for the perfect place
to do some initial setup work.
And it's there for now in this widget
where I want to ask for permissions,
and get the address of this device.
Now we can do all these things
by using Firebase messaging, a new class,
a new object, which we can use,
which is provided here
by the Firebase messaging.dart file.
And I'll just quickly reorder those imports,
but it's this import here that must be added
to use Firebase messaging.
And on Firebase messaging,
we can again use instance
as we did it for all these other
Firebase packages as well
to get access to this Firebase messaging object,
I'll just name it FCM here.
Now on this Firebase messaging object,
we then got various properties,
and methods that we can use.
For example,
we got the request permission method,

and you should call this method first.

Request permission.

Ask the name suggests.

Asks the user for permission to receive,

and handle push notifications.

Request permission, returns a future,

and also takes a couple of parameters

with which you could fine tune

which kind of notifications you want to receive.

Now since this in the end returns a future,

which then in turn yields an object

with more details

about the notification settings,

we wanna await this,

but we should not turn in its state

into a async method,

that's not recommended

because in its state is a method

used internally by flutter,

and flutter does not expect

in its state to return a future.

Therefore, what you should do instead

is add a little helper method,

set up push notifications, could be the name,

and here you can use a sync.

And then we simply want

to call set up push notifications here,

and just not do anything else here.

We can then move this code here

into this set up push notifications method.

And in here we can now safely await calling

FCM request permission.

Now here we'll get back
our notification settings,
and on this notifications settings object,
we got a couple of properties
that in the end tell us
which kind of notifications
can be received by this device here.

Now, here, I'm actually not that interested
in those settings,
but you could fine tune your code,
and the permissions you then sent
to this app here on this device
based on the permissions
that were granted.

Here I'll just continue,
and I'll continue by then calling
get token as a next step.

And that's an important step
because in the end, this token,
which gets token yields is the address
of the device on which your app is running.

And it's this address which you would need
to target this specific device.

So if you have push notifications,
that should target individual devices by address,
this here yields the address
of the different devices.

And you could then send this address
with an HTTP request
to your backend and store it
in a database to connect it
to other metadata of your user for example.

Now get token returns a future,
which in the end yields a string,
and that string will be that address.

So here we get that token
by our waiting get token,
and then here I'll simply print this token.

And with that we should be outputting
that token in the debug console.

As mentioned, you could send this token via HTTP
or the Firestore SDK, for example,
queue a backend and store it
in a database and connect it to user data,
and do anything else you want to do.

Here, I'll just print it.

And now if we reload this app here,
therefore restarts on this Android emulator,
we should be prompted to grant permission
there as well.

So here, I will allow this.

And once we accepted this,
if we go back,
and we open the debug console,
you should see your token there.

If you don't see your token,
simply restart the app.

And now you should be able to see it here.

And this cryptic string in the end
is the address of your device.

In this case,

of this locally running emulator I got here.

28. Testing Push Notifications

Now this token could be used
to then send push notifications to that device.
And to show you a quick demo,
I'll just copy that token here.
And then back in the Firebase project under Messaging,
we can click on Create your first campaign
and choose Firebase Notification messages.
And on this page,
you then can give your message a title like Test
and "Some test message" as a text.
And then choose Send test message.
And here you can paste in that registration token,
so that device token,
the token of the device to which the message will be sent.
Now, before you click test,
make sure that the app is running in the background
by simply going back to the home screen, for example.
Now, if you click test
and you go back to your device,
(device jingles)
you should see that this push notification arrived here.
And if you tap this notification,
the app will be opened and you will be taken to the app.
And that is in a nutshell
how you can send push notifications
to specific devices by using this token,
which you get with help of `getToken`.
Now here we manually copied and pasted that token.
Of course, in reality, as mentioned,
you would send it via HTTP to your backend
and store it in a database there.

And then whenever something happens
that should lead to a push notification
to be sent to a specific device,
you would look up that token in the database
and send that notification programmatically.
And I'll get back
to sending push notifications automatically,
programmatically later.
But that's in a nutshell how you could send notifications.
And this does all not just work on Android.
You can also test this on iOS,
though there, as mentioned,
you will need to use a real iOS device,
for example, a real iPhone to test this.
So here I connected my iPhone to my Flutter project.
And to make sure that running this app
with push notifications enabled on that iOS device succeeds,
there are a couple of steps you should go through
before trying to run this app
on a real iOS device the first time.
In the terminal, run `flutter clean` as a first step.
This will do just what the name implies.
It will basically kind of reset your Flutter project,
of course, without losing your packages or your code.
Once you ran `flutter clean`,
run `flutter packages get`
to reinstall all those third-party packages.
To be safe,
you then also might wanna run `flutterfire configure` again
to make sure your Android and iOS
Flutter Firebase configuration is up to date.
Especially for me here

where I changed this bundle identifier for this iOS project,
this is a step we should not skip.

And now with that configuration updated here,
make sure that in your Firebase project settings
under cloud messaging,

you go to that newly added iOS app
and reload this page if you don't see it,
and upload your key there as well.

So that key file we set up before
with that key ID and your team ID.

So make sure you add that here again.

And then back in your Flutter project
cd into the ios folder and run pod repo update in there.

Then run cd .. to move out of that ios folder
and run flutter build ios to build this project for iOS.

So to not run it yet, but to prepare it for iOS.

And this can take a while.

Indeed, it can take very long, up to 10 minutes or so.

But wait for this process to finish.

And once that build process finished,
make sure your real iOS device
is connected to your macOS device
and you trusted your device on your iPhone.

And then make sure that your iOS device
is selected here as a target device.

And with that out of the way,
you can run this Flutter app on this real iOS device.

And running this the first time can again take some time,
but eventually it should finish

and launch the app on your iOS device.

And then there we of course must log in.

And we then get this permission popup

where we're being asked
for whether we want to allow push notifications or not,
which I will of course allow here.
And if I now send a new notification,
a new test message to that token of my iOS device,
and I click test
and I have the app running in background mode.
Eventually this notification arrives there,
and I can tap it to open the app again.

29. Working with Notification Topics

So now that we got the push notification
basics out of the way,
I wanna adjust the code here for the specific needs
of this demo application we're building here.
Because indeed, with this application,
since we only have one general chat
in which everyone participates,
I don't wanna target individual devices.
Instead, I'd like to have like a channel
to which every app installation subscribes
and I want to be able to send a push notification
to all subscribe devices in one step
without targeting the individual addresses.
And thankfully that is possible as well.
We can use this FCM package,
this Firebase Cloud Messaging package
to call the subscribe to topic method.
And then we can enter any topic name of our choice,
like for example, chat.
Now when we do that,
this device, which has our app installed

and which went through
this Firebase push notification setup process at least once
will receive a push notification
whenever we send a new notification to this chat topic.
Now push notifications can be sent to a topic
from inside Firebase, again by going to the messaging area.
And then there, choosing Create your first campaign
and Firebase notification messages.
And then there,
we can, again, enter some text of our choice.
But now we won't send a test message,
but instead we can click on Next.
And then there under Target, choose Topic,
and then enter your topic name, chat in my case.
If I now click Next,
I can choose when the message should be sent,
and I'll choose Now here.
I don't need any extra settings.
And if I now click Review, I can then publish this.
Before I do that though, I'll quickly restart the app
to make sure that this app installation
really did subscribe to this topic.
And I'll then go back to the home screen here.
And now if I hit Publish,
this message will be delivered to all devices
that subscribed to the chat topic.
Now it can take a second or two for the message to arrive,
but at some point, it should arrive.
And publishing messages to topics is always slower
than targeting individual devices.
But for some apps like this one,
it's the best way of reaching a lot of devices in one step.

So here, the message arrived,
then if I tap it, again, the app opens up.
And this also works on iOS devices.

30. Sending Push Notifications Automatically via Cloud Functions

So now we learned
how we can target individual devices by using that token
or how we can target multiple devices in one go
by using such a topic.
Of course, up to this point,
I always did that manually though.
I sent a test message manually
or I created this campaign here manually.
And for some use cases,
that might be exactly what you wanna do,
but in many cases, this whole process should be automated.
For example, here in this app,
I actually don't wanna send push notifications manually,
but instead, this should happen automatically
whenever a new chat message was sent and received.
So whenever a new chat message was added
to my Firestore database in the end.
Whenever a new document is added to this chat collection,
I wanna send such a push notification to all users.
That could be our requirement for this app.
And to add this kind of automation,
you need some code on your backend,
not some code in your Flutter app,
but instead, some code on your backend,
on your remote backend, which can send out
that push notification to all those devices.
We can't do this in our Flutter app.

We can't write the code for sending out that notification in the app here because we need that backend access.

We need access to the configuration we added here to this Firebase project, and that information and configuration cannot safely be accessed from inside our Flutter app.

Now thankfully, adding code to a backend is quite straightforward.

If you build your own custom backend, you did that by writing code anyways, so then you could add more code to it to also use a backend Firebase SDK to send such a message.

But if you're already working in a Firebase project, it's even easier.

There, under Build, you have this Functions area, which in the end allows you to add code to your backend that can be triggered and executed upon the occurrence of certain predefined events.

Now as you see here, you must upgrade your project to in the end have a paid plan before you can use functions.

Now you will actually be able to get started for free because there is a free tier available, so a certain volume of function executions for which you won't be charged, but you must have a paid plan, you must have a credit card added to your account.

Then you can set up a budget, which in the end simply is a threshold which controls when you'll receive an alert, an email alert once you are approaching or exceeding this amount.

You can learn more about the plan you're signing up to

here on this plans page.

And there you also learn what's included in this free tier.

For example, that for these cloud functions,

you're not going to pay for the first

2 million executions in every month.

Now with that set up, we can use this functions feature,

we can get started with that.

And for this, we'll actually need to install this tool here as a first step.

And this tool in order to be installed,

needs another tool to be installed first.

And that would be Node.js,

which in the end is a JavaScript runtime,

which we don't need for Flutter development in general,

but which we will need for writing this function code,

and most importantly,

for installing that other tool I just showed you.

So on nodejs.org,

you should simply download the LTS version,

whichever version that is

at the point of time you're viewing this.

So click this button, download this version,

and then walk through that installer that gives you.

And once you did install Node.js,

you can copy this command here

and run that on your terminal.

On Mac OS and Linux,

you might need to add `sudo` in front of that.

On Windows, you don't need to add that,

and trying to add it would give you an error.

And here I actually got this accidental

dollar sign at the beginning.

Here we go.

Now that's the proper command.

And this now installs another Firebase tool,

not a tool you need in general

for building flutter apps with Firebase,

but a tool we will need for getting started

with these cloud functions

for adding this backend code to Firebase.

With that tool installed as a next step,

you should go through these two steps

which are presented to you here

by running Firebase init in your terminal,

in your Flutter project here.

And then here you have to choose

which features you want to use.

And even though we are using Firestore here,

we're using it through the Flutter Firebase SDK already,

so here you actually only need to choose functions

by navigating there with the arrow keys

and then hitting space to select it.

You don't need to select anything else,

and you can then hit enter and then choose,

use an existing project to link your existing project

to this Firebase function we're about to add

and select that Firebase project you created before.

And this will now set up this project

for adding this functions code.

And once we wrote a function,

we can then run Firebase Deploy

to upload that function code to Firebase.

But we'll do that later

once we did write that function code.

Now here is part of the setup process.

We're also asked whether we wanna write this function with JavaScript or TypeScript.

And you don't need to know either of these because I will give you the function code.

So here we'll just use JavaScript, we don't need ESLint, so here I'll type N for no.

And we can type Y for yes here to install any extra dependencies that are needed for this function code.

Now once this setup process finished here, you should have a new functions folder in your Flutter project, and you got a couple of files in there.

And it's the index.js file that matters for us here.

This in the end is the file where you will put your JavaScript code, so not Dart code, JavaScript code that will be uploaded to Firebase to run there as a function.

So that's the backend code we can now add to our Firebase project.

And if you would want to write your own functions, you would indeed need to dive into the official Firebase documentation to learn all about this functions feature.

And you would need to learn JavaScript because Firebase cloud functions are written with JavaScript.

But here for this demo app,

I'm providing a finished function to you.

And attached you find my finished index.js file, which should be used to replace your index.js file with.

And this function code I created for you
in the end sets up a function
that listens to a new Firestore document being created.

That's what we're doing here.

And you can learn all about the available events here
with this documentation link.

But this makes sure
that whenever a new document is created on Firestore,
this code here executes.

And then we get access to the document snapshot
that was created,
so to the data that was created in the end.

And we then use the Firebase admin SDK,
which is a tool provided by Firebase
for usage in JavaScript code
to use the cloud messaging service
and send a new notification to a topic.

And here, I'm sending a message to the chat topic
of course on purpose
because that's the topic we're subscribing to
in our Flutter apps.

And then as a next step,
the notification that is sent is configured
by giving it a title that's coming from the snapshot data.

So from the document that was added to Firestore.

And here, I'm accessing the username.

And this code indeed does look quite similar
to what we did here in our Flutter app
when we output our chat messages.

Here, we also worked with that snapshot data in the end
and we access the username at the text like this.

We're doing a similar thing here

and setting the username as a title
and then the message text as a body
and username and text are of course keys
that can be found in Firestore.

So here, if you go to your Firestore database
and you take a look at the chat documents,
you find username and text as keys in those documents.

And I'm extracting those keys here,
and then this setting here simply controls
which internal activity will be triggered in the app
whenever this notification is tapped.

It in the end will ensure that the app is opened.

So that's the code that will be executed on the backend
in Firebase whenever a new chat document is added.

And therefore now we just have to make sure
that this code gets pushed to Firebase
and gets added to their servers.

And to do that, you should open the terminal again
and run Firebase Deploy,
and this will now upload this function code to Firebase
and register it there so that it will be executed
whenever we do add a new document to the chat collection.

So let's wait for this deployment process to finish here.

Here we go, the deployment process finished for me.

And now with all of that deployed, if we restart our app
and we send another message testing cloud functions,
this should now be delivered as a push notification
to all subscribed devices.

And as mentioned before, it can take a short while
before the push notification does arrive
because sending them two topics simply takes a bit longer.

But eventually, it should get here

because of this cloud function that we set up.

And you can go to this functions area

and reload that functions page

to get more information about your function

and to, for example, also see the amount of executions here,

the amount of requests received by that function,

which in the end tells you how often it has been executed.

Now if you're not getting a push notification,

you can also go to your function here

and view the logs to see potential error messages.

And for example, here for me,

I got a status 401 error message,

which indicates that we have some kind of permission error

and indeed, I got a project not permitted error here

as it seems.

To get rid of this error,

you should go to your project settings

and under cloud messaging,

also enable this cloud messaging API Legacy.

For that, you can follow this link here,

which I get after clicking on these three dots

and click enable.

With that enabled, you should now be able

to get those push notifications.

So if I test this again and I then close this app,

this push notification now does arrive.

And that's how you can receive

push notifications in Flutter.

Now, definitely also visit the official documentation

to learn all about receiving push notifications

and what you can configure there

and different ways of handling incoming messages

when the app is running in the background,
in the foreground and so on.
But in a nutshell,
that's how you can handle push notifications
and you therefore have a very broad tool set
that hopefully helps you build your dream app.

31. Module Summary

And that's it for this course section.
Another long course section, obviously,
but a course section where we covered
a lot of crucial topics.
We did not just practice building another app
and using more widgets,
but instead you learned how to integrate Firebase,
a service offered by Google to add a dynamic backend
to your Flutter app,
and how you can use that backend
for sending push notifications.
You learned how you can handle these push notifications
in your Flutter app,
and of course, you also learned how you can use Firebase
for authenticating users, for storing extra data,
for uploading files to Firebase,
and how you can then use those files,
how you can display those images in your app,
and you learned how you can build this chat app,
how you can send those messages to Firebase
and store them there,
and how you can make sure that your chat app
and your screen automatically updates
whenever new messages are received.

This hopefully is helpful to you
and helps you build any kind of app you wanna build.