Firebase & AI Chatbot

Transkrip referensi — folder sumber `4_flutter_ai-chatbot-n-firebase`

14 bagian · dibuat otomatis

Daftar Isi

1_getting-started-with-flutter-and-firebase

1. Course Overview and Learning Objectives

Hello everyone.

Welcome to module one course Overview and Setup.

In this video we will introduce the course and discuss a crucial concept in flutter development state management.

Managing state efficiently is essential for building responsible and scalable applications.

And today we will explore why it matters and how the provider package simplifies this process.

So let's dive right in.

Why state management matters state management is a crucial concept of Fletcher app development, especially

when dealing with real time data.

The provider package is one of the most popular and efficient way to manage state in flutter applications.

So what is a provider?

Provider is a simple flutter package built on top of inherited widget system.

It simplifies state management by making it easy to share and manage application state efficiently across widgets.

Provider ensures efficient state updates and scalability, making apps more responsive.

Now moving on.

The key benefits of provider reactive updates widgets automatically rebuilds when state changes.

Scalability.

It means it's suitable for both small and large applications and separation of concerns, which keeps the UI and business logic separate.

Now how provider works provider separates the business logic from the UI by creating state.

State is held in a class that notifies listener when it changes.

Second step is providing state.

State is provided to the widget tree using change notifier, provider or other types.

Third is consuming state widgets.

Use consumer or provider to listen to and rebuild based on state changes.

So let's quickly go over what we've learned about introduction of provider.

In flutter provider is a simplified state management, making it easier to share data across widgets.

It uses change notifier to trigger updates and ensure widgets reactivity.

It maintains a clean and scalable architecture, keeping UI and business logic separate.

And that wraps up today's discussion on provider in Fletcher.

Thank you for watching.

In the next video, I will show you an example of how to implement it.

2. Introduction to Flutter and Firebase Architecture

Hello everyone.

Welcome to module one course overview and video two.

Today we will explore the overview of Flutter and Firebase.

In this session we will dive into how Flutter and Firebase work together to create seamless scalable applications.

So what is Fletcher?

Fletcher is Google's open source UI toolkit designed for building beautiful, natively compiled applications across multiple platforms.

Which Fletcher you can develop apps for iOS, Android, web and even desktop, all using the dart programming

language that enables efficient and reactive app development, making flutter a powerful choice for motor applications.

So why should developers choose flutter?

First, Hot Reload accelerates development by instantly reflecting code changes, allowing for quick iterations.

Secondly, it provides a highly customizable UI and ensures smooth performance.

Finally, flutter compiles directory to native code, ensuring fast execution and seamless user experience.

Now let's talk about Firebase.

Firebase is Google's backend as a service that simplifies app development by offering cloud based tools.

It provides backend services like authentication, real time databases, cloud messaging, and more, eliminating the need for complex backend development.

This makes Firebase an ideal solution for mobile and web applications.

Firebase comes with a range of powerful tools, including authentication and real time database authentication

for secure user access, and real time database for dynamic and synchronized data update cloud messaging

and advanced analytics.

Cloud messaging for push notifications, enhancing user experience, built in analytics to track user behavior and optimize app performance.

Scalable and secure backend services.

It is used to handle all your app's data needs.

So why should you use Firebase for your app's backend?

First, it manages backend infrastructure, handling server management, hosting and scaling so developers can focus on building features.

Second, it reduces development complexity for prebuilt services, eliminating the need for custom backend solutions.

And most importantly, Firebase allows developers to focus on the front end and core business logic, ensuring a smooth user experience.

How to Flutter and Firebase work together.

Flutter seamlessly integrates with Firebase through the official plugin, making it easy to add backend functionality to your app.

The Firebase plugin allows seamless integration of backend services.

It supports user authentication, managed cloud database and send push notifications, all with minimal effort.

When we combine Flutter and Firebase, they offer several key benefits.

First, it provides a unified experience across multiple platforms with a single codebase.

Secondly, it simplifies backend development, reducing the need for complex server side coding.

And lastly, it enhances full stack app development, making it easier to build feature rich applications with minimal effort.

So to summarize, Flutter and Firebase together provide a robust solution for app development, while Firebase handles backend operations effortlessly so your app becomes efficient and scalable.

Together, they allow developers to build efficient, scalable and cloud connected applications, all with less effort and more efficiency.

Thank you for watching.

I hope this session gave you a clear understanding of how Fletcher and Firebase work together.

3. Installing and Configuring the Flutter Development Environment

Hello everyone and welcome back.

In today's video we will talk about creating new project and setting up Firebase.

In this video I will show you how to create a new flutter project in Visual Studio Code and connect it to Firebase.

We will start by setting up a new flutter project, selecting the right options and organizing our files.

Then we will connect our project to Firebase step by step making sure everything runs smoothly.

Next we will install Firebase CLI login and initialize Firebase for our app.

Finally, we will ensure everything is configured correctly for smooth development.

By the end you will have a fully setup flutter project with Firebase integration.

Let's get started okay so to create a flutter project we're going to go into view and click on Command Palette and New Flutter Project.

And we're going to click on application.

It's going to ask for our destination where we want to save the project.

So you can select any folder.

I'm selecting a folder in my C drive.

And next I'm going to name my project.

And now it's going to work on getting all the packages and every setting for my project automatically.

So now that the project is set up, let's move on to connecting Firebase to it.

So here we're going to create an account on Firebase.

And then you're going to name your app I'm going to name it Flutter Course.

And continue.

Okay.

So now that it's finished creating I'm just going to continue it.

So now that the project is created you're going to get an option to add an app here.

And you can choose to create any app.

I'm going to click on flutter because we're working with flutter app.

So the first thing you're going to need to work with Firebase is install the Firebase CLI for your computer.

So you're just going to have to go ahead and do that first.

And for installing CLI you're going to need MPM as mentioned here.

So you're going to have to download that first.

And you can just download it from here.

So installed is downloaded and set up on your computer.

You can open a command prompt.

And in here you're going to type npm v and it's going to tell you if it's installed or not.

Then you're going to have to run a special commands.

Which is this one.

You're going to copy and run this into your terminal and it's going to globally activate CLI on your computer.

So I'm just going to run this.

So apparently it's already set up on my laptop.

So I don't need to do this.

But it should work properly if it's not already installed.

So the next thing you're going to need to do here is you're going to run another command.

Just Fletcher login.

So you just go to s code and just paste it here.

So let's just say enable for now.

Now and I'm just going to allow it.

And using my account.

So our Firebase CLI was logged in successfully.

So let's close these tabs.

And I'm going to click next.

And I'm going to copy this command to activate CLI globally on my computer.

Okay.

So now that CLI is activated I'm going to go back to my Chrome browser.

And I'm going to copy the second command.

And this command will be used to connect our project.

So now that is connecting directly to my project.

It's going to ask you which apps you want to choose to create.

It.

And I'm going to choose just Android and iOS for now.

But if you want to make it multiple on multiple platforms, you can choose web, windows and Mac OS also, so you can use spacebar to unselect or select these and just enter it.

I'm going to go back to Firebase.

And from here I'm going to start on my next step which is to initialize Firebase.

So for that you have to copy this command and go to your library and Main.dart file.

And you have to paste this here the other imports.

Copy this and paste them here right below our other import.

Then you have to go to.

Dev.

And here I'm going to search for.

And here you're going to search for Firebase Core.

And I'm just going to copy this from here and go back to VSCode.

And in my perspective YAML I'm going to paste this under dependencies and flutter.

So let's save this file.

And once that's command is finished running I'm going to go back to my main.

And let's just add an async here because we're awaiting it.

And there you have it.

Flutter and Firebase are set up in our project completely.

And you can connect and start running your apps quite easily.

So in conclusion of this video we've successfully created a flutter project and connected it to Firebase.

We've learned how to set up the project, organize files, install Firebase CLI and integrate Firebase into flutter.

Now your app is ready for features like authentication, database or cloud storage.

Keep experimenting, explore Firebase features and build amazing flutter apps.

Thank you for watching and see you in the next video.

4. Creating a Flutter Project and Connecting Firebase

Hello everyone.

Welcome to module one Course Overview and Setup.

In this video I will guide you through setting up flutter on your windows laptop.

We will break it down step by step.

First we will download and install flutter from the official website.

Next we will choose the right setup for windows.

Depending on what you want to develop mobile, web or desktop apps.

Then we'll configure the environment variables and important step to make sure flutter runs smoothly

on your system.

We'll verify the installation using flutter Doctor a command to check if everything is set up correctly.

By the end, you will be all set up to start building flutter apps.

Let's get started.

So the first thing we're going to do is go to Google and we're going to type download lecture here.

Then we're going to go to the docs and we're going to install it manually.

So from here you have to select what system you're going to be using.

I'll be using windows but you can choose Mac OS, Linux or Chrome whichever suits you.

And then from here you can install the latest version if you want.

You can also install older versions from here, but I'm just going to go and recommend the latest version because they're the most stable.

And then I'm going to wait for it to download.

Okay.

So now that's finished downloading I'm going to go to this folder, and I'm going to copy it from here.

And then I'm going to go into my local disk and I'm just going to paste it here.

Okay.

So now that it's finished copying I'm going to go to my flutter folder and into my bin folder.

And I'm going to copy this path from here.

And then I'm going to go to the search bar and search for the edit environment variables.

In the advanced tab we're going to go to the environment variables.

And here in user available for admins I'm going to go to the variable.

And here is supposed to paste there.

And then we're just going to okay this.

Finish it up.

Then I'm going to search for command prompt.

We have a special command for checking if ledger is available on our system.

I'm going to do Fletcher doctor.

So as you can see we have the Fletcher installed successfully.

Visual Studio Android studio are not directly related to Fletcher.

These tools we use to run our Fletcher programs.

So there is not a problem.

So next let's check our Fletcher version.

And as you can see we have our Fletcher version which is the latest 3.35.6.

And that's it.

You successfully installed and setup Fletcher on your windows laptop.

So to summarize what we have done in this video, we have installed Fletcher on our windows laptop.

We went through downloading and setting it up, then configured environment variables to ensure everything

runs smoothly.

Finally, we use Fletcher Doctor to verify the installation, making sure everything is ready to go.

Now you're all set to start coding.

Thank you for watching.

I hope to see you in the next video.

2_flutter-basics-project-setup-and-widgets

1. Flutter Project Structure Overview

Hello everyone and welcome back.

Today we will be covering an important topic setter project structure.

Our focus will be on understanding the flutter project structure, its key components and how to organize files efficiently for a well-structured application.

The first thing we'll be doing is understanding how flutter project being organized is crucial for efficient development and maintainability.

A structured approach helps developers navigate, scale, and manage projects more efficiently.

Second is the importance of organizing files and directories.

A well-organized project improves workflow, enhances collaboration, and makes debugging easier.

Proper directory management ensures a clean and scalable codebase.

Third are key components essential for flutter development.

We will explore important components like the lib folder which contains the core code, the asset folder for media and resources, and platform specific directories.

Our fourth point is understanding dependencies and project setup.

Managing dependencies is a crucial part of flutter development.

The pub spec YAML file helps add external packages that extend project functionality, making the development

process more efficient rather than just exploring theoretically.

I'll now walk you through the actual project folder in Visual Studio to show you how these components are structured in real projects.

Let's get started.

So the first thing we look at when we're talking about the structure of a project is the name.

The name needs to be of a standard.

It can be either lower snakecase, upper snakecase or anything like that.

Try to make it conventional so it's easier to find projects that tools and ideas are more of binary

files, so we do not have to make any changes in this.

Here you find app Gradle and the necessary file that you will use to make changes for your Android.

After that we have the iOS folder.

And in iOS you're going to find components like flutter, runner and Info.plist file, which you need to use to make some changes in your iOS apps.

Next we have the lib folder, which is the core of your flutter projects.

This code is where all the action happens.

You'll use it to write your flutter codes.

Assign screens and implement the main functions.

The project starts with the main screen, which is initialized using the main and runner, which is a material app.

Inside Material App, you have theme data in which you can set your color schemes and everything for your entire project.

For now, I've created a screen inside my main class, but that is not the right way to do it.

For that, you usually have to go to live and you create a new folder for screens, and then you paste this code into another file that you create in the screens folder.

Other than that, you can also create a models folder where you will store your models.

Next we have our perspective YAML file, also known as the yml file.

This file is where you add the dependencies for your project.

I'm sure you remember we've added our Firebase Stalker functionality, which we use to connect to Firebase.

To add a package, you can press Ctrl Shift plus P and you can just choose a package.

From here you can go to Chrome and search for Pub Joseph.

Like for instance we searched for Firebase Core and it regenerates Firebase Core.

We copied it from here and we added it below our dependency.

Finally, once everything is finished you can run the app.

So for that you have to close your yml file, save your project and go to run and start debugging.

Or you can also choose to run without debugging.

Then it's going to ask for the device you want to connect.

I'm going to deselect my device, but you can also choose to run it on Chrome.

And these are your debug controls.

You can choose to pass a running project, restart it, or stop it entirely.

And that's all for this video.

So to wrap things up, here are some conclusion and key takeaways of today's lecture.

A well organized flutter project ensures efficient development and maintainability.

A clear project structure improves efficiency, reduces complexity, and makes it easier to scale the application.

Secondly, key directories like lib acids, Android and iOS play essential roles in structuring the app.

Each directory serves a unique purpose.

Lib for core functionality.

Assets for media and platform.

Specific folders for Android and iOS.

Integration.

Perspective is crucial for managing dependencies and external packages.

Dependency management plays a vital role in extending Flutter's functionality through third party packages.

Proper folder segmentation screens models.

Utils improved modularity and scalability using well structured folders keeps the code base clean and helps developers manage components more effectively.

Always verify package compatibility across different platforms before integration, ensuring compatibility across Android, iOS, Linux, macOS, and windows.

Prevents unexpected issues during development.

That's it for this session on Flutter project structure.

I hope you have a clear understanding of how to organize your project for efficient development.

See you in the next video.

2. StateFul and Statless Widget with Example

Hello everyone and welcome back.

This video we will be looking into stateless versus stateful widgets in flutter.

Today we will dive into the difference between these two widgets with example and demonstration.

Before we begin, let's quickly go over the key topics we will cover today.

We will start with an introduction to the widgets.

Next we will explore stateless and stateful widgets with examples.

Then we will compare their key differences, how they manage state when they rebuilds, and their impact on performance.

Finally, we will conclude with the importance of understanding widgets, ensuring you know when to use stateful and stateless widgets for efficient app development.

Now let's get started with an introduction to widgets in flutter.

Let's start by understanding what widgets are inserted.

Widgets are the fundamental building blocks of the user interface.

They represent everything you see and interact with on the screen.

Plato categorizes widgets into two types stateless and stateful widgets.

Moving on to stateless widgets, a stateless widget is a widget that does not require a mutable state.

Once it's built, it remains unchanged during runtime.

It is best used for static components without needing to update based on user actions or data changes.

For example, I am going to demonstrate a text widget for this example.

First, we will extend state class to stateless widgets.

That means that the widget class does not maintain any mutable state.

So now that we have our main file, I'm going to go to the screens folder and I'm going to create a new file here.

I'm going to call it our text example dot dart.

And let's make this a stateless widget.

I'm going to rename it text example.

Build method is the core of every letter.

He describes how to display the widget in the UI.

Lastly, we have text widget.

Text widget is used to display the message.

In our example, our message would be hello, I am a developer.

This is the content that will be shown on the screen when the widget is rendered.

So here I'm going to replace the placeholder with a text widget directly.

And our text is going to be I am a developer.

And I don't think I need to do is go to your main class and in your home section you add the name of the file you just created.

I'm going to name it Text Example here, and now it's going to reference my text example file.

As you can see, I am saying the text I am a developer on the app.

Now let's look at stateful widgets.

These widgets have mutable states.

This means they can change their properties or appearance during runtime.

This is achieved with the help of an associate state object.

A stateful widget is ideal for dynamic components that need to rebuild based on user interaction.

For instance, a button that updates a counter value.

In this example, we are going to extend stateful widgets to state class, which means that each widget can hold mutable state.

The state class state is responsible for managing the state of the state class widget.

Now, to make this into a stateful widget, I'm going to remove this code and add stateful layer.

I'm still going to name it text example.

It holds immutable state such as the counter in this example.

Then I'm going to create a counter here.

I'm going to assign it value one.

Then to show this in a text widget I'm going to show this in a column.

And text.

You don't need to understand the column widget.

Now, I'll explain it better.

So the text will be counted.

And this is how you show a dynamic variable.

Add a semicolon here and save our file.

As you can see, we are seeing the value of a counter dynamic.

Next we will add the elevated button widget.

This is used to trigger actions.

In this case, when the button is pressed, it calls the counter add method to increment the counter value.

Next, let's add an elevated button here so the user can dynamically update the value of a counter.

Add the chiles.

I'm going to say add text.

Click me.

Finally we will add the counter add method.

This is responsible for incrementing the counter variable and then calling Setstate.

The Setstate method triggers the UI to reveal the updated value of the counters.

Then for dynamically updating here I'm going to create a method.

Counter L.

And here I'm going to say counter.

Plus this.

And then I'm going to set state.

And then on our on pressed I'm going to add here counter add.

Let's save this file and check out our emulator.

Once we click on our click Me button the counter is updating it dynamically before the user.

The state is updating using the Setstate method.

And that is how we differentiate stateful and stateless widgets.

Here is the detailed comparison between stateless and stateful widget.

First aspect is state.

Stateless widgets have a fixed state.

Once they create it, they do not change.

While stateful widgets have mutable states, they can change dynamically during runtime.

The second aspect is lifecycle.

A stateful widget is only built once and will only rebuild if its parent widget changes.

In contrast, a stateful widget can rebuild any time when the Setstate method is called.

The third aspect is performance.

Stateless widgets are lightweight since they do not need to manage state, making them more performance

efficient.

Stateful widgets, however, are slightly heavier due to the state management.

To wrap up, let's quickly summarize the key takeaways.

Stateless widgets are ideal for static components like text and icons.

They are simple, efficient, and do not require state management, making them great for fixed UI elements.

On the other hand, stateful widgets handle dynamic and interactive elements.

They allow updates based on user interaction or data changes, making them essential for features like

counters, patterns or animations.

Thank you for watching and see you in the next session.

3. Text Widget in Flutter

Hello everyone!

Today we will explore the most commonly used widgets in flutter.

In this session we will cover the essentials of Flutter text widget.

We will explore how to customize text styles such as color, size, and weight.

We'll discuss the importance of using scaffold for basic app layout.

We'll learn about safe area that ensures that text and content are within the screen.

Safe boundaries.

And we'll also look at how handling text overflows ensures that no content gets cut off on different screen sizes.

And then we'll also discuss the conclusion summary of it all.

Moving on to the introduction, flutter offers a wide range of widgets to help build user interfaces efficiently.

These widgets are highly customizable and are essential for creating interactive applications.

One of the most commonly used widgets in flutter is text widget.

E.g. it allows developers to display a readable text content on the screen.

In this session, we'll explore various ways to customize text in flutter, including changing color fonts and alignments to improve the overall design and readability.

Next is the scaffold and safe area.

Scaffold provides a basic structure for the app layout scaffold, which acts as the foundation of the app providing structure.

It includes elements like app bar, body, and floating buttons, ensuring a well-organized interface.

Safe area ensures content is displayed within safe boundaries.

The Safe Area widget prevents content from being overlapped with system UI elements, such as notches or the status bar.

It ensures a clean and accessible layout.

These are used to structure and display text widgets properly.

By using both scaffolds and safe area, we ensured the text widget is displayed properly without layout issues.

Improving user experience.

Moving on to the basics of the region.

The widget.

The text widget in flutter is the safest way to display static and dynamic text on the screen.

A basic implementation of a text widget looks like this.

Text and then Hello world or any other string.

This displays the text exactly as written within the quotation marks.

It accepts various style parameters for customization.

Letter allows extensive customization using textile.

We can modify properties like font size, weight, color, letter spacing, and text decorations to enhance readability and design.

So let's get started.

So first thing I'm going to show you is what the screen looks like with and without a scaffold.

So for now I'm just going to show this text widget.

And I'm just going to put it down as style text save this and run it.

So this is what the text will look like without any scaffold.

So if I remove this text from here.

Add a scaffold and in the body I add my text.

Now let's check it out.

As you can see, it looks much better with the white background and the text looks also a bit user friendly.

So now I'm going to start styling the text.

For that I'm going to use the style property.

And against that we add the text style.

First up I'm going to increase the font size to 20.

And as you can see it's a bit more visible now.

And I'm going to change the color to color start blue.

Then I'm going to increase its font weight to make it bold.

And you can see it's a bit bolder.

You can also see it's difficult for us to see the actual text because it's appearing right under our status bar, and the time and our text is overlapping.

So for that you have to wrap your scaffolds control dots wrap with widget a safe area.

Save this.

And now you can see the text will automatically appear under the status bar.

So to continue our styling I'm going to next add a font style and it's going to be font style dot.

Italic.

Next is the decoration property.

Usually we use the decoration property a lot because it has most of the features for decorating text.

For now I'm just going to use it for text decoration underline.

And then for letter spacing I'm going to say space it by two and space the words by four.

Save this.

And you can see the text is italic Balls and underline and all the properties we applied to it.

Now, as you know, overflowing is a main issue in flutter.

So if I were to have a really long text say is large, it will take 1 to 2 lines or more.

Then H.

And let's also increase our font size to say 40.

I'm going to make it's max lines to two.

So if it overflows I want it to wrap on to the next line.

So save it.

And as you can see, instead of overflowing like it usually does, it will go into the next line automatically.

Alternatively, if you want to show it on just one line but you don't want it to overflow also obviously.

So you're just going to use overflow property and you're just going to say text overflow dot ellipsis.

And instead of wrapping to the next line, it's just going to show three dots and just trail off from there.

And that's it.

That's all you need to know about these basic widgets in flutter.

And that wraps up our demonstration of flutter widgets.

So to conclude this session, the text widget is flexible, allowing developers to control text appearances through styling options.

Using textile.

We can modify text size baldness, make it italic, underline it, or change its color, making the text visually appealing.

Other than that, with safe area and scaffold, we can ensure that text is displayed correctly without being hidden by the system UI elements.

Flutter provides properties like max, line and overflow to control text display, ensuring that long text does not get cut off unexpectedly.

And that brings us to the end of this discussion.

Thank you for watching and I'll see you in the next video.

4. Container Widget in Flutter

Hello everyone!

Today we are going to explore the container estate in Fletcher.

In this video we will talk about the key features of the container widget.

Next is its implementation.

We will look at how to apply these features.

And finally we will discuss its conclusion.

So moving on to its key features.

Versatile layout widget.

It provides an easy way to structure layouts and handle positioning, making it a go to widget for flutter developers.

Styling capabilities.

One of the main strengths of the container widget is its ability to style UI elements.

Interactivity.

The container widget also enables interactivity, creating a more engaging user experience.

Now let's move on to the implementation of the container widget.

So I'm going to start by adding a container to the property of my scaffold.

And the first property of the container is going to be the child.

You can assign this anything a text or another container maybe, or an expanded widget.

So literally anything goes.

So I'm going to give it text and login button.

And you can just see that this is just text because you haven't given it any color or shape as of yet.

The next thing I'm going to add is the color.

So color is dot amber.

Save it.

And you can see the background color has changed.

And the rest of the decorations you have to do in the decoration property.

So box decoration.

Now we have a problem here.

If you add the color and decoration at the same time, you're going to get an exception that says you cannot have both of them at the same time.

So what you have to do is move the color property inside of the decoration to avoid that exception.

Next I'm going to add a border and I'm going to say border dot all.

And I'm going to give it a color.

The color starts black and I'm going to set its width to one.

Let's save it.

You can see we have a little border.

Let's increase the size of this container.

For that you have the height property and the width property.

Save this.

And you can see now that we have our container.

It looks much better.

Now we have these sharp edges.

And that does not look appealing in any UI.

So I'm going to add a border radius.

Border radius dot circular.

I'm going to give it a value of ten.

And let's just save this.

And you can see it looks much better now.

We have another problem here that our text is stuck to the top left of the container.

So to make it a bit more padded I'm going to use the padding property.

I'm going to say inside stores.

Only and from top I'm going to give it 20.

You only use the agent sets dot only property when you want to give specific value.

Otherwise you can give the all property and just pass one value.

So our text is appearing somewhere in the center.

Let's just increase it from the top here.

So another thing is that our container is stuck against the top corner of this screen.

If we want to give it a bit more margin from here, we're going to use a margin property.

And I'm going to say edge inserts dot all and use ten.

And there's going to be a bit more difference between the container and the edges of the screen.

Another property we have in this is the box shadow property.

You have to pass an array.

So I'm going to say box shadow color.

I'm going to give it color or black.

And I'm going to say blur.

The radius by ten.

Let's save this.

And now we have a very dark shadow around the container.

Now to make it look more like a button, I'm going to change this to 15 and change the height to 50.

Let's save it.

And now it looks more like a button, right?

So I'm going to wrap this container control George with another widget called the gesture detector.

And on tap if the user taps on this entire container I want you to perform this functionality.

Now this can be any functionality here.

Um, let's say we add a log that says button pressed.

Right.

And.

Add a semicolon here.

Save it.

Now, if I click on the login button you're going to see it's sending our action which is button pressed.

So you can perform any functionality here.

For now I've just added a logo and that's all for this video.

So in conclusion the container widget is essential for creating customizable UI elements in flutter.

It is one of the most commonly used widgets, offering great flexibility in design.

With the container, you can customize the design by using box decorations for borders, gradients, and background images.

Padding and margin help manage the internal and external space around the container, while box shadow adds depth to the widget by detecting user gestures like tap or long press, you can make your UI more engaging by combining styling options and interactivity.

And that's all about the container widget.

Pleasure.

Thank you for watching and I'll see you in the next video.

5. Expanded Widget and Flexible Layouts

Hello everyone!

Today I will walk you through the expanded widget in flutter.

In this video first we are going to talk about expanded widget in flutter and how it creates responsive UI.

How to apply it effectively in your layout and understand how it helps distribute space efficiently.

Next we will look at its implementation in layouts like rows and columns.

Finally, we will conclude with why it's essential for dynamic adaptable flutter interfaces.

So let's get started.

This widget is essential for managing space dynamically in layouts.

It helps to prevent overflow issues in rows and columns, with expanded widgets automatically adjusting their size instead of being constrained by fixed values.

Moving forward, let's see how we can use expanded widget in flutter.

First, we have our containers with the expanded widget.

This fixes any overflow issues that might occur due to fixed sizes.

Where's the flex property to distribute space dynamically?

This works in both horizontal and vertical column layouts.

Here's where we shift to the actual implementation where I demonstrate this step by step.

So I created some boilerplate code so we can build on that.

I'm going to grab my container inside of a row.

By the way.

You can just press control Dot and it will give you suggestions.

And then I'm just going to copy this container.

And paste it.

And to differentiate between the two I'm just going to make this one blue.

So let's save this file and run it.

So as you can see it's overflowing from the right side here.

If I go ahead and remove the width constraint from both these containers.

Save it.

You can see that now just adjusting to the text within them.

So what will happen if I wrap them in an expanded widget?

So I press control and Dot and I go to wrap with expanded.

Control and expand.

Let's save this.

And as you can see, they are dynamically adjusting their width to take up the necessary space without actually overflowing or strangling upon each other.

And if I remove my margin property from these containers and save it now they are next to each other without the extra padding between them.

Let's remove the transform property.

Save this.

As you can see, there is no more space left between these two widgets.

Regex.

When working with the expanded widget, you also can use another property that these have, which is flags.

If I put this flags as one, nothing's gonna happen.

But if I put it at two and save this, it's going to take up more space as compared to the other one.

Similarly, if I increase it to three.

It's going to take up one third of the space.

And if I copy this expanded and create another one here, save it.

It's still going to take as much space as it's possible in a one third.

And the rest are going to divide the space between them.

Next I'll modify the flex property to let's say back to one, and we're going to have them dividing the space equally among them.

The same goes for columns if I remove this raw property.

From here, save it.

These containers are lining up vertically instead of horizontally, and they're taking up half of whatever space is available.

Similarly, you can update the flex property of any of these.

Let's say I'm going to do the middle one.

And as you can see, it's going to start taking up more space as compared to its other sibling.

And that's it.

That's all you need to know about the expanded widget to wrap things up.

Here are some key takeaways from this session.

The expanded widget helps create responsive layouts by distributing available space dynamically.

It prevents issues caused by hardcoded height and width values.

It is useful in both row and column layouts.

So that's all for today.

I hope this helps you understand how to use an expanded widget effectively in flutter.

Thank you for watching and I'll see you in the next video.

6. CircularAvatar, SizedBox and Divider in Flutter

Hello everyone!

In this session we will explore some essential widgets in flutter.

These widgets help create clean, structured and visually appealing layouts in mobile applications.

I will break down our discussion into the following sections.

First, we will look at the introduction to these widgets.

Next we will discuss its key properties.

Then we will dive into its implementation and how to use them.

Finally we will talk about its conclusion.

Moving on to the introduction.

What is the circular avatar widget?

It is a built in widget used for displaying profile pictures or icons in a circular shape.

Common use cases in flutter.

It is commonly used in user profile sections, messaging apps and dashboard.

Importance of size box and divider for UI structuring size ensures proper spacing between UI elements

while divider visually separates content sections.

Its key properties setting radius, and background color.

We'll start by defining the avatar size using radius and customizing it with the background color,

adding an image to the avatar instead of a solid color.

We can display an image using a network or asset image.

Using sized box for spacing.

Size box allows us to manage spacing effectively between widgets, keeping the layout clean.

Implementing divider for UI separation.

Finally, we will add a divider to create a clear separation between sections.

Let's now move to the practical side where I will demonstrate how to implement these widgets.

In a step by step.

I'm going to start by adding a center widget here, and to each child I'm going to assign a circle avatar.

And now you can also set its background color.

I'm going to set it as colors dot Ember and save this.

You can see a small circular image color.

And then I'm going to add a background image which I'm going to set as network image.

And you can paste any random URL here I just copied a random image from the internet.

Let's save it.

And now we can see the image.

Now if the image is not rendered, it will just show an ember background.

Now to increase its size, I'm going to set its min radius to 50 and max radius 200.

Save this.

Now you can see now the image is up to 100 radius.

Now what if I want to show two circular avatars vertically?

I'm already using a column, so it shouldn't be any problem.

Let's save this.

Now you can see they are very close to one another.

For that, you have to add a divider here.

Let's use the divider widget.

Set its colors to say colors dots black.

And then I'm going to set its thickness to two and the height to two.

Also, let's check it out.

If we want more space between the widgets we can always use a side box.

So size box of height 100 here.

And let's copy this.

And one after the divider also.

And now you can see there are very visually centered.

And there's a lot of space between each of them.

And that's it.

That's all you need to know about a circular avatar and a divider.

You can set the color and the size box for which you can set either the height or the width, depending on if you're using a column or a row.

Finally, moving on to the conclusion, the circular avatar makes it easy to create visually appealing profile images with customization.

Size box helps maintain consistent spacing and improves layout structure.

The divider enhances readability by neatly segmenting different sections of the UI.

By combining these widgets, we can create clean, structured and user friendly applications.

Circle, avatar, sidewalks, and divider are essential elements for creating visually appealing and well-structured letter applications.

These simple yet powerful widgets play a key role in UI design.

Thank you for watching.

I hope you're implementing these widgets yourself so you can practice them and get better.

7. TextFormField Widget

Hello everyone.

In today's topic, we're going to learn about the text form field widget in flutter.

We'll start by understanding what the text form field widget is.

Next we'll see how to set it up with the controller decorations and keyword configurations.

Finally, we'll summarize some key takeaways.

Now that we have an overview, let's begin with the introduction.

What is the text form field.

It's a widget in flutter that allows user to input text.

It's commonly used in forms, search bars, and login screens.

You can validate input and try to match your app's theme.

Forms in flutter rely heavily on this widget for handling user input.

We can use a controller to manage and retrieve user input.

The controller is connected to the fields for real time text handling.

We can add labels, borders, and styles to improve usability.

We can also set the appropriate keyboard layout based on the expected input.

Now let's move on to its implementation.

So I'm just going to create a text form field here.

And the first thing I need to provide to this field is a controller.

So for that you have to create a new variable where you can store your text data.

So I'm just going to call it for example I use a controller.

Make sure you follow the right naming conventions.

For example for the file names we use Snakecase.

But for our variables we tend to use lower camel case.

And I'm just going to pass this controller here an extra controller property.

Next let's start with the decorations into a text form field to pass input decorations.

And I can set a label text here.

It sort of like a watermark test that just shows before user has entered anything in there.

So I'm just going to say enter your email for example.

Let's save this and see what we have so far.

You can see we have our text form Home fields where we can enter data.

So let's add a border first.

So it's properly defined in the UI.

So outline input border.

Save this.

And now you can see it has a border.

And here is our label.

You can also enter a hint text.

Let's say I'm going to give the hint text the same.

You can make it different if you want.

Just email or mail here.

Save this.

You can see the hint text shows here while the label text shows on the top, and when you try typing, the index goes away.

Okay so next we're going to add an icon.

So at the start you can say a prefix.

And I'm going to pass it an icon.

An icon is basically an image that's built in in our packages.

So pricing.

And here you can see an image right next to your text.

Let's wrap this in a padding widget and check it out.

This looks very much like an input you would see on a login page.

Here you can input any information that you want, and here you can also add a keyword type.

So I'm going to say text input type dot email address.

Save it.

And on my emulator it's not showing but usually it shows a special type of keyboard when you're trying to type in the field.

You can also choose from a variety of other input types.

Let's wrap things up with a key takeaway text.

Field is a versatile widget.

It provides a powerful way to handle user input in flutter apps.

Essential for handling text input in forms, whether for login screens or full forms.

It makes text handling easier.

It can also be customized, including keyword type for better input handling.

You can adjust its appearance and set the right keyboard for better usability.

That's it for this session.

Thank you for watching and I'll see you in the next one.

8. Stack Widget in Flutter

Hello everyone!

Today in this video we will be diving into the stack widget in Fletcher, a powerful tool that allows you to layer multiple widgets on top of each other.

Let's look at the overview of the stack widget.

The stack allows us to layer multiple widgets on top of each other, making it perfect for creating complex UI layouts and visual effects.

Next, we will work through the implementation of the stack widget.

I will show you how to add widgets, position them, and adjust the alignment to create dynamic layouts.

Finally, in the conclusion, we will recap how to stack widget helps you layer elements and manage UI elements flexibly within the flutter application.

Starting with the understanding of stack Widget.

Flutter stack widget allows us to place widgets on top of each other, unlike rows or columns that arrange them side by side or vertically.

It is useful for layered UI elements such as text or images, floating buttons, or overlapping designs which follow the order in which they are declared, meaning the first widget appears at the bottom and the last one is on the top.

Now let's see how this works in flutter by implementing it.

So let's add a stack widget in my column.

And it's also going to have a list of children.

I'm going to first create a container.

Let's give it a face height and width.

Let's say height of 100 and a width of 100.

And I'm going to give it a color as red, say red.

Next I'm going to create another child.

It's going to appear on top of my container.

If I place it before my container, it's going to hide behind it.

So let's say.

You can see it's appearing here.

And since it's default alignment is top left, it's the text is appearing on the top left.

By default, the height and width of the stack is the size of the largest child.

So currently both the child and the container are aligned at the top left.

And if I change the alignment and make it let's say center.

So alignment center.

The text is also going to appear in the center.

And if I change it to let's say bottom right, the text is going to appear in the bottom right.

Now if I remove my text widget and create another container and I'm going to give it the color blue.

Let's see.

And as you can see, the red container which is completely swallowed by the second container which is appearing on top.

If I reduce its height and width let's make it 90 or 80.

You start seeing your original container.

So the main thing about stack is their children appear from bottom to top.

So whichever widget you want to show at the top is going to have to be the last widget.

I.e. if I want to make a text widget appear at the top, I am going to add it in the absolute end.

And you see it's going to start appearing in the.

And if I delete it and add it in the middle, it's going to start hiding behind my containers.

And that's it.

That's the basics of the stack.

We also have the position property in the stack.

But we're not going to go into that right now because it's a bit more complicated.

So to summarize it all, stack layers widgets from bottom to top, following the order in the children's list.

Alignment properties like alignment center position widgets within the stack.

The stack widget allows us to layer widgets, control the alignment, and create visually engaging layout.

That's all for this video.

Thank you for watching and I'll see you in the next one.

9. ListTile Widget in Flutter

Hello everyone!

In this video I am going to introduce you to an essential flutter widget that is widely used in app development, which is the widget.

We began by understanding the purpose and structure of the widget.

Each tile is a multifaceted widget for structured list items.

It supports layout with the title, optional subtitle, and a leading and training widgets.

It is perfect for displaying user profiles, setting options, and navigation lists.

Now that we understand the widget, let's jump into the implementation and build a structured UI using that.

So I'm going to start by creating a list style widget in my column.

And the first property we're going to use is the title property.

It's going to be bigger than the other ones and more bold.

So I'm going to use it for my main text which will be user profile.

Then I'm going to add a subtitle to it.

And this is going to appear below my title.

And it's going to be smaller in size.

I'm going to add text View or edit user profile.

Let's save this and see how it turns out.

As you can see, we have our data with the title being the bigger heading and the subtitle being the smaller one.

So let's add an icon to the leading property.

You can pass it anything.

Let's first try adding an icon.

Icon search person.

And now you can see there's also an icon next to our text.

Next I'm going to replace this icon with a circular avatar which we've already explored in our previous video.

I'm going to give it a background image and I'm going to pass it a network image, any URL here.

As you can see, our image is appearing right next to our text.

And another property in our style is the trailing property.

You can add anything here also because you can see it do accepts a widget.

For now I'm going to pass it in.

I can I can start let's say fork Right.

Let's add a comma here.

Save this.

Now you can see this looks more appealing.

You can use it on your settings page or anywhere else you want to have your profile picture.

And then your title and subtitle explaining where you're going to be navigating.

And the actual icon buttons for navigation.

And the most important property of the title is its own tab property.

You can add anything here for when the user clicks on the title.

You can add a log or a navigation or anything else you want.

And that's it.

That's all you need to know about the style.

Now that we've implemented the list style widget, let's quickly recap what we've achieved today.

We successfully structured our list style with the profile image as the leading element.

We added a title and subtitle for informational displays.

Then we included an actionable trailing icon for navigation.

Additionally, we added interactivity using quantum function.

This simple yet powerful widget enhances UI clarity and functionality in flutter applications.

Thank you for watching.

I hope this tutorial helped you understand how to use the List Style widget effectively in your projects.

10. ListView Builder in Flutter

Hello everyone!

In this session we will be exploring another essential flutter widget.

The List View Builder further provides a powerful widget called List View Builder, which is used for dynamically generating list items.

Instead of creating a fixed number of list items manually, this builder creates them on demand, making it highly efficient.

One of its key advantage is performance optimization.

It renders only the items currently visible on the screen, reducing memory usage.

This makes it essential when handling large data sets, ensuring smooth scrolling and efficient resource management.

Now that we understand why it's important, let's explore how to implement it effectively.

So usually our data is coming from a backend of some sort such as Firebase.

Or it can be SQL and we do not know the number of items we are going to receive.

So for that we usually go with our list view builder.

It doesn't require us to tell it how many items is going to need, because if I want to create multiple styles but I do not know how many they're going to be, I'm just going to go ahead and use a list view builder.

Let's first return a text widget from here, and it's just going to print the current index.

So index is index.

And let's tell it manually for now how many items it should count.

So let's say I'm going to count it 100.

And I'm also going to wrap this in an expanded widget because it's going to be scrollable.

And list views inside a column always need to be in an expanded widget.

Otherwise it throws you an error.

So let's save this.

And as you can see, it is counting our index starting from zero going up to 99.

So 100 in And the best part about this is you can customize it however you like.

So I'm going to replace my text widget with my list style widget, which you created in a previous video.

I'm going to catch it from here and add it here.

Save this.

Now you can see our UI looks much more appealing, and it's going to print 100 instances of the same list type.

If you want, you can make it dynamic, but that depends on what your needs are.

Next, let's try to make it a bit more dynamic.

I'm going to create a list of strings and I'm going to call it list for now.

And I'm going to assign it a couple of names.

So John.

Jane.

Adam.

So now that we have a list, I'm going to go to my item count.

And I'm going to say don't.

Instead of manually calculating it.

I'm going to say follow the list's length.

However many items are in my list, just print that many and in my user profile.

Instead of printing just this manual string, I'm going to print instead the names.

So I'm going to say list of index.

Let's save this and see if it works.

As you can see, it's printing the correct names and that's how you make a list view dynamic.

So in this tutorial we learned how to use List View Builder to create dynamic lists, customized list items, and optimize performance.

To wrap up, this video is an essential widget for handling dynamic lists efficiently.

It ensures smooth performance by rendering only the items that are necessary.

Debugging common issues, such as missing item counts and layout errors further enhances its readability.

Thank you for watching.

I hope you got a clear understanding of the list and how to efficiently use it in your app.

3_firebase-authentication-basics

1. Firebase Setup and Autheniaction setup

Hello everyone and welcome back.

In this video we'll be talking about setting up Firebase Authentication in our apps.

Let's quickly go over what Firebase is and why we are using it in our app.

So what is Firebase.

Firebase is a backend as a service platform providing authentication databases and cloud functions while integrated with flutter because it provides seamless back end integration for cross-platform apps.

What will we set up?

Fibrous cord that connects Flutter to Firebase and Firebase authentication which enables user sign in with email and password.

Now that we have an understanding of Firebase, let's move on to the practical implementation step by step.

So I'm going to teach you how to set up Firebase Authentication in your app.

And from the console you can see that it is properly set up for my project for both Android and iOS.

So I'm going to go into the build section.

And from here we're going to go into authentication.

I'm going to go ahead and click setup Sign in methods.

And for now I want to just use email and password signing.

You can choose from a variety of options such as Google, Facebook, Apple and others.

And you can also choose anonymous.

But for now I'm going to choose Email and Password.

And in this you also have two sections.

You can choose to log in using email and password.

Or you can either send user a just an email link.

Whenever they enter the email, you just send them a link so they don't have to remember the password for login.

So for now I'm just going to use email and password.

And I'm going to save this.

And that's it.

That's how you just easily set up Firebase Authentication to support for your project.

Now I'm going to go to my VS code.

And in here I'm going to add a new dependency to support my Firebase authentication.

So just control shift P add dependency.

And I'm going to search for Firebase auth.

So it will add it successfully.

Now if you go into your Gmail file you should be able to see Firebase auth under dependencies right here.

And that's it.

That's all you need to do to set up Firebase Authentication in your project.

Now that we have successfully set up Firebase Authentication in our project, let's summarize what we accomplished today.

First Firebase is now fully integrated.

Our flutter apps is successfully connected to Firebase allowing it to interact with various Firebase services like authentication and databases.

Next authentication is enabled with implemented Firebase authentication which lets users sign in securely using email and password.

And finally, Firebase services can now be expanded easily with Firebase properly setup, adding features will be much smoother.

Thank you for tuning in and I'll see you in the next video.

2. Adding images in your Flutter Project

Hello everyone and welcome back.

In this session we will learn how to add images to our project effectively.

First, we will organize our images inside an asset folder to keep our project structure.

Next, we'll set the correct image paths for imports to ensure our images are loaded properly.

Finally, we'll demonstrate the final UI with images added.

Now that we have.

Now that we have an overview, let's move on to the practical implementation of adding images to our project.

Okay, so I want to add images to my project.

For that you can go to this one.

You can drag and drop your images right inside your flutter project, which will be a mess because you're going to have more than one image in your project, obviously, and that's just going to clutter your main files.

The other, more preferred approach is that you create a new folder and genomic assets folder, and within that folder you can add all your images, logos, icons, everything.

So even the asset folder is usually structured to hold more images and logos in a better format.

So you create subfolders for it.

For now I'm just going to use the assets folder, and then I'm going to create a new folder in it called the images folder.

And I'm going to copy my images from here that I'm going to use in my project.

And I'm going to go ahead and paste this in my images folder.

And I'm just going to say paste it.

And now if you open them you're going to see images right inside your folder.

So now to tell a further project where these images are located, I'm going to go into my YAML file and under the shade line for build your assets here I'm going to add assets.

Next line are you going to add a hyphen.

And then you just present the path.

So assets slash images because that's the part where our images are located.

I'm going to save this file.

Now it's going to start importing the images.

Next I'm going to create a new file in my project.

I'm going to call it the home screen again for that.

And let's call this Statefulwidget images also work in stateless widgets.

Let's replace this with a container.

And I'm just going to add an asset image as a child.

So for using the image in the app you're going to use image dot asset.

And now here you just refer to the path to your image.

So assets images and then whichever image you want to select from here for example you want to do this one login logo JPEG.

So you have to name it exactly the same.

So login logo George same extension PNG.

Let's save it and run this file.

So as you can see our image is being displayed successfully.

This completes the image insertion process for our UI.

To recap, we learned how to correctly structure and import images in a flutter project.

We also saw how keeping our images inside an asset folder improves organization.

Lastly, we viewed the final UI preview with images successfully added.

Thank you for watching.

Keep experimenting with images to enhance your app's.

3. Building a Sign-In Form in Flutter

Hello and welcome to this tutorial, where we will be creating a sign in form from scratch in Flagyl.

In this form, we will write through the process of creating a user friendly sign in form in Flagyl.

First, we will focus on adding the essential text fields for email and password input.

Next, we will make sure the input fields are styled properly, ensuring a clean and visually appealing design.

Finally, we will implement a functional submit button to handle form submission effectively.

All right, now that we know what we're building, let's move on to the implementation.

Okay.

To start creating a sign in Chrome, I'm going to first replace the placeholder with a scaffold.

I'm going to set the body property.

And I'm going to assign in the column widget and its children.

First thing I want to show on my form is an image.

So I'm going to use the size works to set its height and width.

I'm going to get the image from assets.

Assets images and blog in local start page.

Next I'm going to set its feet property to box fits church cover so it doesn't get cut.

Next add some text.

Let's call it sign in.

And I'm going to set its style property to textile.

Set its font size.

I want it to be a bit bigger.

So mine is 32.

And on to it will be one or both.

And lastly the color I want to be black.

So color starts black.

And you can see we are seeing our image and text rendering perfectly.

Now I want to do is show it in the middle of the screen.

So I'm going to say main axis alignments dot center.

Next I'm going to wrap my column with a padding widget and set padding dot all to 24.

And finally also wrap our column with a single child scroll view for smaller screen.

So the screen is smaller.

It will be scrollable instead of just text being cut off.

Now I will use will automatically start rendering in the middle because we've set this property.

Next I'm going to set.

A size box because I want some space between my widgets.

I'm going to say it's 530.

Then I'll use a text form field for input.

And I'm going to set its decoration space.

So input Decoration.

I'm going to send the text to email.

I want the fields to be true.

And fill color will be colors.

George.

Gray.

George white.

Opacity.

Opacity for now.

And white is 0.25.

And finally I'm going to set the border outline input.

Border offset is border radius.

So border radius or circular.

And I'm going to set it to 0.25.

And border size I want to show none.

So I'll just select that.

Next select the prefix icon.

And again.

I can short showed email and color.

I want it to be black, so I installed black.

Finally, I want to set the keyboard type.

So I'm going to set the text input type dot email address.

And next we're going to create a controller for it.

So text engine controller email.

Controller is equal to text editing controller.

And then I'm going to copy it and create another one for our password field also on the gateway.

So password controller.

And then we just have to pass it to our controller property.

Let's save and check it out.

As you can see it's appearing here.

Now the radius is a bit less.

So I'm going to replace it with 12.

And that looks much better.

Next let's create another text fields.

So I'm just going to copy this.

We can put a height of 20 and replace this with the password controller.

Index is also going to be password.

And that's pretty much all we're changing on our password fields.

So I'm going to set it to raw log.

And then from password field I'm also going to be adding a suffix icon basically a trailing icon.

And I'm going to be wrapping it in a gesture detector.

So when the user taps on it I want the password to be visible or hidden.

So for that using gesture detector and the icon make it a static icon.

For now we'll change it.

And it's color to color black obviously.

Now we have our password skills.

Next I'm going to add another text.

Before that we add the size rules for spacing.

And I'm going to create a row of it.

So children and set its.

Main axis alignment to main axis alignment ends.

And the first thing I'm going to be adding to the row is a text fields for get password.

I'm going to style it a bit.

So style and textile I'm going to add its color to colors, Dodge orange to go with our theme.

And I'm going to set the font size to 14.

The decoration color will be color orange also.

And finally the font weight will be font weight dot.

Specifically w 700.

So it's a bit less bother than the actual word property.

Next, I'm going to create a size box for a height of 32, and I'm going to be creating another one

its width to double to infinity.

So it will take as much space as possible.

I'm going to create an elevated button, and I'm going to set it on price to nothing for now, but we'll

be replacing it later.

Let's replace this child with arrow.

Also, I'm going to set its image to assets and six images.

Human eye can image dot PNG another size box for spacing.

I'm going to set the weights to ten because it's a row and not a problem.

And then I'm going to set the text as Booker sign.

Okay.

And I'm going to change the row properties to an exercise.

I'm going to set it to main first.

And then I'm going to set main access alignment to main access alignment set.

Next let's add some styling to our button.

So for that I'm going to also use its style property and elevated button dot style font.

I'm going to set the padding to as inside the matrix.

And let me set the vertical to 16.

Then shape it rounded rectangular Triangular border and set the border radius to border radius dot circular

and send that to trial.

Finally, let's also add its color.

I'm going to set the background color to color orange and the foreground color will be color white.

This is the background color property.

Now you can see our Google sign in button.

Next let's also add a sign in button.

I'm just going to copy this size works Ctrl C and add it here.

I'm going to remove this text property from here and replace it with the row.

And I'm going to remove the Google sign in with the regular sign.

And let's increase textile.

Form size is going to be 12.

I check it out now.

You can see our sign in button is visible also.

Finally let's create another size block height 20.

And the last going to be adding here is don't have an account.

So I'm just going to create an inside arrow.

Main X is methyl center.

And children first we have the actual text.

So don't have account.

And I'm going to style it a bit textile.

Color color swatch black.

And I'm going to set the font On size two.

Next I'm going to create text that will be clickable so we can navigate to another screen if we want to using that.

But for now just keep it empty and I'm going to assign it a child of text.

Sign up.

In this trial.

Color is going to be color shot orange.

And font size is going to be 14.

And then the decoration to text decorations dot underline.

Decoration color is going to be color orange.

And finally the font weights with the font weight Dot 700.

So let's save this.

And now we can also see our don't have an account text.

And let's say we created a very useful UI for taking input of user email and password for sign in.

To conclude what was done in this session, we started by adding text fields for user input and use text editing controllers to manage and retrieve enter values to enhance the UI.

We applied input decoration, adding hidden text borders and colors to make the fields more user friendly.

For the password fields, we enabled obscure text to hide the input, ensuring security.

We also set the keyboard types to optimize user experience.

Finally, we added an elevated pattern to handle form submission, making the signup process interactive and functional.

And that's it!

We successfully built a simple yet effective signup form in flutter.

Thank you for watching and I'll see you in the next video.

4. Building a Sign-Up Form in Flutter

Everyone.

In this video, we will be setting up our sign in screen with navigation and sign up.

In this video, we will see how to enhance our sign in screen by adding a sign up option.

We make the sign up text tappable using gesture detector.

Then we'll setup navigation with navigator dot push and remove until to smoothly transition to sign up screen.

This ensures a seamless loop for users moving between sign up and sign in screens.

Alright, now that we know what we're doing, let's jump into the implementation and see how to bring this to life in flutter.

So I've created this UI for my sign in screen and we have this button for don't have an account sign up.

So we're going to make that available.

Let's scroll to the sign up section.

And I'm going to wrap this text with a gesture detector.

So control dot rapid widget gesture detector.

And I'm going to create its own tab function.

So now what we want to do is navigate to the sign up screen.

So for that we're going to have to create the sign up screen.

So new file.

I'm going to call it sign up screen.

To start I get a stateful widget.

Now we can go back to our main screen.

For signing in navigator Dot, push and remove one cell and to the new road.

You're going to pass the material page route and to the builder you pass the context.

And for navigation I'm just going to use an arrow function for the sign up screen.

And to the route I'm going to say Route Rabbit.

And then I'm going to set it to false.

And next I'm going to create my sign up screen using the same UI as my sign in screen.

So I'm just going to copy the scaffold.

And on my other screen I'm just going to replace it here.

And I'm going to remove our don't have an account logic because we don't need it here.

I'm going to copy the controllers for email and password and paste those here.

And remove the unnecessary return statement.

And since we're signing up on the screen I'm going to replace sign in strings with sign up.

So let's say whatever files.

And restart our program.

I should go to sign in screen where it changes navigating to our sign in screen again.

Instead of sign up.

So let's replace that with sign up screen and rerun our program.

Now when I click on sign up it's going to navigate to the sign up screen.

And as you can see our sign up navigation is working perfectly.

So to summarize what was done, we made a sign up text available and linked in to a new screen using gesture detector.

We also used navigator, dot, push and remove and seal to ensure smooth and intuitive navigation.

We copied the scaffold from the sign screen and adjusted for the sign up screen.

We've successfully added a sign up option and smooth navigation.

Thank you for watching and see you in the next one.

5. Sign-Up Firebase Function

Hello everyone!

Our topic today is Firebase Signup function.

We start by setting up Firebase Authentication in our project and configuring it to allow email and password signup.

Next we'll write a function that interacts with Firebase to register new users with their email and password.

Then we'll implement error handling to catch issues like weak passwords or already registered emails and provide user friendly feedback.

Finally, we'll integrate the Firebase function into our app's UI so users can easily create an app.

So let's move on to the implementation.

So now that we have to create an odd service for signup, I'm going to create a new folder for services and a sub folder specifically for auth services.

Auth services.

And in this folder I'm going to create a new file called the OTT services file.

I'm going to close.

I'm going to create I'm going to create the class auth services.

And in this I'm going to be creating a static function that's going to return a future of string.

And I'm going to name it sign up with email.

We're going to passing it two parameters email and passwords of string type.

So wrap this in a try catch block.

So if any error occurs we can handle it gracefully.

And for the firebase auth dot instance dot create user.

And you just pass it the email and password parameters.

And it's just going to try and login the user from here.

Next I'm going to return a string.

In both cases of success or failure.

So sign up successful.

And if it fails, I want to return another string.

And this.

That's going to be error during sign up.

And I'm just going to interpolate it with the actual error message.

So either toString.

Now we're going to need two more functions here.

One will be showing a Snackbar.

This is the widget we'll be using to show the information if the login was successful or not.

So I'm going to be passing it a string message and a build context.

So.

Final Snackbar variable is equal to Snackbar widget.

And the context will be passing a text widget message that we passed as a parameter, and then we'll start styling it using the style, property and textile.

I'm going to set its color to colors dot white, and I'll also be setting its background color to colors dot orange.

Add a semicolon and I'm going to set its behavior to snack bar behavior dot floating.

And I'm going to set its duration to let's say duration of seconds and three.

Now all you have to do is show the snack bar.

For that we use scaffold messenger dot of we passed it the context.

And then we just say show snack bar.

And to this we pass our snack bar variable and that's it.

Now to actually show the message from our sign up with email, I'm going to create another function and I'm going to name it handle.

Sign up.

It's going to be requiring all the same parameters plus a build context.

So password and build context.

Let's pass the context.

And I'm going to first retrieve the message by calling our.

Signup function.

So I'm going to say sign up with email.

And then all I have to do is pass it to my show Snackbar function.

And that's it.

Now I've made it super usable.

I can use it with any other function.

Now let's head over to our sign up screen.

And in my signup budget I'm going to start calling the methods we created.

So auth services dot handle signup and I pass it the email controller Text.

You don't pass it the entire controller, you just pass its text property.

And then I'm going to pass it the password controller dot text and make sure to convert them to string in case they are just integers.

And the semicolon.

And that's it.

That's all you have to do for your front end.

Let's go to sign Up screen and enter our email.

So you can enter any email here.

Next I'm going to enter a password sign up.

And as you can see sign up was successful.

So to conclude we successfully implemented Firebase.

Sign up with email and password.

We created a reusable function to handle authentication making it easier to integrate anywhere in our app.

We also added error handling and feedback ensuring smooth user experience.

Thank you for watching this video and I'll see you in the next one.

6. Sign-In Firebase Function

Hello everyone!

In this video we will explore Firebase Sign-in function.

We'll learn how to implement the sign in function using Firebase Authentication.

We'll go through the steps to write the function connected to the sign in screen, and handle errors like incorrect credentials.

By the end, you will know how to authenticate users securely in your app.

Let's get started.

Okay, for handling sign in, we are going to go to our All services class first, and we have to create the same functions as sign up.

So I'm just going to copy this.

And I'm going to change this into sign up with sign in.

Same for sign in here we're replacing our create with email and password to sign in with email and password.

Now I just have to change my strings to sign in so we know if there are any issues in our sign in.

And that's it.

That's all you need to do to change the back end.

Now going back to our UI, I'm going to navigate to my sign in button at the very end.

And here on the I'm going to add my auth services dot handle.

Sign in.

And I'm just going to enter the email controller.

Dot text and password controller dot text.

And that's it.

Now this semicolon here I save this.

And one more thing we need to change in our auth services is in our handle sign in.

I'm going to sign in with email instead of password.

So let's reload it.

And let's add our email and password.

The median and then bracket sign it.

And as you can see sign in was successful.

Also in the future we're also going to navigate to our home screen or dashboard screens.

But for now we've completed sign in successfully.

To conclude what we've learned today, we successfully implemented Firebase Sign In.

In our app, we connected the function to the sign in screen and passed email and password from text controller.

We also handled authentication errors to improve the user experience.

Thank you for watching and I'll see you in the next video.

7. Loader-Added on SignUp and SignIn

Hello everyone!

In this video, we'll learn how to add a loader to the sign up and sign in screens in ledger.

Let's quickly go over by adding a loader is important.

A loader enhances user experience by providing visual feedback during authentication.

It prevents confusion by indicating that the app is processing the request.

The loader appears only when needed.

When the sign in or sign up process is in progress.

This ensures a seamless and responsive integration into our app.

Now that we understand the importance, let's move on to the implementation.

So let's start by creating a boolean variable for loader.

So bool I'm going to name it loader.

And initially its value is going to be set to false.

Then where I'm calling my function, I'm going to say set state and set the value to true.

And also after it's done, set the loader value to false and to show the actual loader, I'm going to go to the chai property and I'm going to make it condition.

So loader for the condition directly after the question mark is when loader is true and after the colon is when loader is false.

So for when loader is true I'm going to create a center widget.

And Jewish child I'm going to set it as circular progress indicator.

And I'm going to set its color to colors dot orange.

Let's set the weight here to tell the function to wait while this method is running.

And I'm going to create a new account So you can see our loader is showing and the sign up is successful.

Now if I create it again loader is still showing.

And it tells us that since we've already created an account with this email, we can do it again.

Now that it's working on sign up, let's go back to our sign in screen.

I'm going to create another boolean loader here and set its value to false.

And then where we're signing up I'm going to create Setstate.

And I'm going to set the loader value to true here.

And the same after the function is finished.

So loader is equal to false and await this function.

So basically the setstate does that.

It tells the UI that we need to perform some changes on some part of the UI doesn't tell specifically.

So the entire widget tree is rebuilt.

This part entire is rebuilt.

And then that shows our loader.

So let's replace it from here.

Also loader.

If it's true, create a center.

And then set its child to secular progressive decorator.

Color is going to be colors.

Dot.

I recall in here.

Let's save this and check if it works on our sign in form.

Let's try signing in.

Sign.

And as you can see, sign is unsuccessful and we could temporarily see our loader.

And on a side note to hide our password text from showing.

I'm going to go to the password fields and search obscure text to true and do the same on our sign up screen also.

So let's just copy this.

And in our password field I'm going to just set it to true.

Now if we enter any text it's going to be automatically hidden.

And that's it.

That's all you need to know about how to show a loader and make the UI responsive so the user is not confused.

Now that we've successfully added a loader, let's recap the key takeaways.

The loader provides clear feedback when user signs up or signs in, it disappears automatically once authentication is complete.

This enhancement improves app responsiveness and overall usability.

It's a simple yet effective way to enhance the UI.

That's it for this video.

Thank you for watching and I'll see you in the next one.

8. Firebase Google Sign In Function

Hello everyone.

Today our topic is Firebase Google sign in function.

In this video we'll learn about Google Sign in function and how to integrate it into our flutter app using Firebase authentication.

We configure Firebase to support Google authentication.

Finally, we'll implement the sign in functionality and test it.

By the end you will have a working Google sign in setup in your flutter app.

Let's move on to its implementation.

So the first thing we're going to need to add our Google sign in support is go to Firebase Console.

And under authentication.

And in the sign up method I'm going to add a new provider Google.

I'm going to enable it.

Next I'm going to configure my email address and save it.

Let's finish this.

And I'm going to go into my project settings and under The certifications.

We need to add our Android and iOS fingerprints here.

For that, you have to go to your flutter project and in your terminal I'm going to paste this command.

It is very specific.

And it's going to provide me my SSH keys one and two.

56.

So the command is basically the path to your Java.

And then there's a bunch of other keywords.

So if you do not know the path to your Java, you can simply go to a PowerShell node command prompt or PowerShell command.

And here you're going to run this command.

And this is going to find your Java path for you.

Like here is mentioned my Java path.

So I just copied that and run it.

And it's going to get it for you.

And then you can just add it to your path and run this to get your saves.

So you just copy your one key first.

You paste it and fingerprint and it's going to ask for the keys, paste it and it will know already which one it is.

Save this one and then go ahead and plug the other one of the terminal.

And just add fingerprint.

Remove the spaces there and save it.

So now we've successfully added our SSH keys.

I'm going to go ahead and download the Google Services dot JSON file.

Next we're going to go to your downloads folder and copy this file.

Then open your Android in the File Explorer and inside your Android and app folder you paste it.

So I already have it, but I'm going to replace it since I've added the new SSH keys.

Once that is finished, You can go ahead back into your project and command shift P.

You're going to add a new dependency.

Google.

Sign in and just add that to your project.

Next I'm going to go into my auth services class.

And in here I'm going to create a new function make it static.

And it's going to return a map of string and dynamic values.

And let's make it nullable.

I'm going to call the function sign in with.

Google.

So the first thing I have to do is create a Google sign in account variable.

I'm going to call it Google User and I'm going to await it Async.

And then I'm going to use another variable that I'm going to create in a minute.

It's going to be called Google sign in dot sign in methods.

So let's go ahead and create it now.

Static final Google sign in and let's call it.

Sign in.

And assign it a Google sign in object.

So after creating this instance I'm going to go back to my function.

And I'm going to check if my Google user, Google user is null.

Then we're going to be returning null from here.

Otherwise I'm going to create a Google sign in Education.

And I'm going to call it Google Earth and Google User dot authentication.

Next I'm going to create and or credentials.

And I'm going to pass it my Google Auth provider dot credential and bundle up my credentials and bind them to access token Google apps.

Dot access token and pass it the ID from my google.id token.

Next I'm going to create an instance of user credential.

Let's call it that.

And I'm going to pass it Firebase auth instance and sign in with Credential and you just pass it your credentials.

And now we're going to log it.

So log user credentials dot two string.

Import it.

And then I'm going to grab this with a try catch block.

So we can know if there's any error and it gracefully.

Google sign in error and show whatever the error is.

And we can also return null from here.

And also here.

So we've covered all our bases.

Next let's go to our sign in screen and where we're logging in using Google.

I'm just going to do auth services dot Sign in with Google.

Let's save this and run our app.

Sign in using Google.

Just enter your email.

Next.

I'm going to enter my password.

In just next.

And I'm going to verify it from my mobile device.

And finish setting up.

And as you can see we're successfully logged into our Google account in our app.

So to conclude in this video we've successfully added Google Sign in in our flutter project, implemented the sign in functionality, and we tested the full authentication flow.

That's it for this video.

Thank you for watching and I'll see you in the next one.

9. Creating Forget Password Screen

Hello everyone!

In this video we will learn how to create a forget password screen in our app.

This feature allows users to reset their password securely if they forget it.

Enhancing both accessibility and security, we'll be creating a password reset screen for users to enter their email when they tap on the button.

Finally, we'll integrate its functionality, which is implementing email based password reset using Firebase Authentication.

It handles the reset process efficiently, ensuring a smooth user experience.

By the end of this video, you will be able to implement a fully functional password reset feature in

your app with the overview complete.

Let's move on to the implementation.

So let's create our forget password screen.

Forget password.

I'm going to make it a stateful widget And replace our placeholder with a scaffold.

Let's start by creating the body.

So I'm going to assign it a single child scroll view.

So it's scrollable.

And then I'm going to assign it a column.

I think I'm going to create in this is an asset image.

So let's go with a path to that.

Copy it and replace this.

Copying the path makes it easier because then we can ensure that there are no actual typos.

And set its width to 250.

Also.

And the weight will be box filter color.

Matching this.

So let's head over to our sign in screen and grab our text with a gesture detector widget.

And on tab.

I'm going to create a navigation navigator dot push and I'm going to add another package add dependency

and it's called Create Transition.

Then I'm going to use it in my navigator transition.

Import it.

And then I'm going to assign it a child which will be forget password screen.

Then I will be signing in the type parameter which is required.

And I'm going to say page transition type dot fade.

Okay.

So let's restart the program.

Now if I tap on my forget password screen it navigates us to our forget password screen.

Next I'm going to copy this text and this text field so I don't have to style them independently.

So copy this.

And in my under my seat I'm going to paste this here.

Just replace sign in with forget password and also create a controller.

So text editing.

And let's check it out.

I'll never forget password screen.

As you can see it's rendering.

We need to add a bit of padding here.

So I'm going to go ahead and grab my column with a padding widget and assign it ten and check it out.

It looks much better now.

Next I'm going to copy my elevated button from here.

Let's copy that.

And I'm going to add it below my text form field after size box.

Remove the on tap functionality here because we're not going to be using the same kind.

And I'm going to change the text to send password reset email.

We do not need to style it here right now, so I'm just going to remove this and add my button in the size box.

I am going to set its width to infinity so it takes up maximum space.

Let's save this and check it out.

Now you can see our button text is working.

Next I'm going to start working on my own press function.

Here the first thing I'm going to check is my email controller.

Dot text dot is empty.

If so, when the user clicks on the button without entering an email, we want to show a message saying that this is not a valid action.

You need to enter an email.

So I'm going to use show Snackbar and the text will be please add email.

Keep it short and simple.

Else I'm going to check if the email is valid.

For that we have this function that checks if the email is valid using a regular expression, which basically checks using a series of expression if the email matches them.

So for example, it has to have an advocate symbol and it has to have some text before and after and

a dot and after that it can be.org.com things like that.

So for that I'm going to just check it here.

Pass the email controller dot text here.

I'm also going to show a snackbar.

So let's just copy this one here and replace the text with.

Please enter a valid email.

Last is our else case where we'll be performing our actual forget functionality.

So for that I'm going to go to my auth services class.

And here I'm going to create my function for checking that.

So future and it's going to return nothing and the name will be reset.

Forget password and email.

And I'm going to be passing it two parameters.

One will be the string of email and the other will be a build context for showing the snackbar.

So let's wrap this in a try catch block.

And in the try module I'm going to add an await first.

And then I'm going to use my Firebase auth class instance dot it's function and password reset email.

And just pass it the email.

And then I'm just going to show a snackbar here.

And the message will be reset password email sent successfully.

And the semicolon.

And then I'm going to show another snackbar here for if we are unable to do it.

And I'm going to say error in reset passwords and just show whatever the error is and just add a semicolon.

So let's save this file and go back to our forget password screen.

I'm going to add the auth services dots resets forget password I'm going to send it the email controller

text and the context.

Next I'm going to create a loading state here also.

So bool is loading equals false.

Let's check that here.

Said state is loading.

Set it to true.

Otherwise you set it to false Is false.

And let's await our function.

So it actually shows the loader.

So let's start this account.

Let's create a new account first so we can test the entire flow.

And set any random password here.

Sign up.

Sign up as successful.

Let's go back on our sign in screen and click on Forget Password here.

I'm going to enter my email.

Send an email.

It seems we forgot to add a note here so let's add that.

If it's not a valid email then you go into there and show the snackbar.

Let's try this again.

And as you can see email was sent successfully.

Let's go on our email.

And as you can see we're receiving our email link.

Let's click on that.

And if I change my password, save it.

Go back to my app.

Now I'm going to go back to my sign in screen.

And I'm going to try signing in using my new email and password sign in.

And the sign is successful which means we successfully reset our email using Firebase.

So to conclude what we implemented today, we successfully built a fully functional forget password feature.

We ensured email validation and error handling for smooth user experience.

We securely manage password resets to Firebase authentication.

Now why does it matter?

The feature enhances both security and user experience.

It allows users to recover their account easily while maintaining security standards.

A well implemented password reset system makes an app more professional and user friendly.

That's it for this video.

Thank you for watching and I'll see you in the next one.

4_firestore-database-and-storage-crud

1. Home Screen and TabBar

Hello everyone, and welcome back.

In this session we'll be talking about home screen and using grab bars for navigation in project.

So now that we've learned all the basics of flutter, let's move on to creating a working app so we

can learn to implement them in real life.

We'll be creating a poll app, which is basically used for creating polls and having users work on them.

The first thing we'll be doing for that is creating a home screen.

Next, we'll use a table view for navigation, allowing users to switch between tabs seamlessly.

Finally, we'll be adding a floating action button so users can easily create a new poll with a single

tap.

Alright, now that we have a clear understanding of what we're building, let's jump into the implementation.

So let's start working on our home screen.

I'm going to go in my features folder and create a new folder in there.

I'm going to call it the home screen.

And create a new class in each arm.

Or screen.

And make it a stateful widget.

Rename it.

So usually we replace our placeholder with a scaffold.

But when we're working with tab bars we replace it with our default tab controller.

And I'm going to set the length static by three.

And you can make it dynamic if you want.

And to each child I'm going to pass it the scaffold and a semi-colon.

And then I'm going to start working on its app bar.

A bar is basically something that shows on the top of the screen.

You can show your title or icons in it.

For example, I'm going to add a title const and a text widget.

I'm going to pass it Text create poll.

Then I'm going to make sure the title is centered.

So I'm going to use the center title property.

And then we have something called the actions property.

And you can pass multiple icons in this or any other widget.

I'm going to use an icon button for now.

And set it to nothing for now.

But I'm going to replace the icon with icon and I can change settings.

If we want to create a settings screen later on, we can just use this.

And then I'm going to use its position property.

And I'm going to pass it my tab bar, which is going to be the tabs at the top of my screen.

And I'm going to be passing it a list of tabs.

Since I mentioned they're going to be three, I'm just going to add three tabs, add some text to it and just do the same two more times.

So I'm just going to copy this.

Let's name the first one all and I'm going to name the second one close to it, and the third one voted for showing the polls that have been voted on.

Next, I'm going to set some of its property for decorations, such as the indicator size.

I'm going to set it to tab bar.

Indicator size.

Dots.

Tab.

Then I'm going to set the indicator colors to color swatch orange going with our app theme.

Next will be the unselected label color.

And I'm going to set it to color short black label will be our tab text.

Next I'm going to set my label style.

Textile for busy and I'm going to pass it the color.

Orange.

I'm going to set the font size to 18 and the font weight to font weight dot 600.

And you're going to have to set the screens for the tabs you just created.

So obviously I'm going to need at least three tabs.

Um, so I'm going to set them below my app bar.

For that you use the tab bar view property and you pass it a list of children.

Of course, because we have multiple tabs.

For now I'm just going to pass it containers.

But later we're also going to set up the individual screens.

And each child is going to be text and just make it the same text for now as the chair.

And let's just copy this.

So finally I'm going to add a floating button in my directly inside my scaffold outside my body.

And I'm going to pass it the floating action button and set the onpress press you nothing for now and

to his child.

I'm going to assign another icon so I can I can store it at.

So let's save this and then I'm going to update my home main for now to directly take me to the home

screen.

Save this and let's run our program.

So as you can see now we have our basic tabs and we can navigate between our screens.

And then we also have our action button.

So that's it for the home screen.

We're going to later start working on power Creations and working on board functionalities etc..

So to summarize what we've done today, we've created our home screen and we use Tab to navigate between

the multiple screens.

Finally, we implemented a visually appealing design.

So that's it for this video.

Thank you for watching and I'll see you in the next one.

2. Create Poll Screen - Part I

Hello everyone and welcome back.

In today's session, we will be learning about creating a pole screen.

We'll start by building a pole creation screen where users can create and manage poles to handle different

views.

We'll design an interactive form using text form fields where user can input the pole title and options.

We'll use box decoration for structure and box shadow for subtle visual depth.

All right, now that we have a clear understanding of what we're building, let's jump into the implementation

okay.

So I'm going to start by creating a pole creation screen in my home screens folder.

I'm going to name it Create Pole Screen.

With that I'm going to make it a stateful widget and rename it to Create Pole Screen.

And next I'm going to replace my placeholder with a scaffold widget.

Appbar bar will be a bar, and I'm going to set its title to Text Widget and Create poll.

Next I'm going to make the title Property to center.

Let's start working on the body property.

I'm going to assign it a padding widget and add insets dot call.

And I'm going to pass it 16.

And then I'm going to set each child as a column widget.

And children.

Let's create a text form field and I'm going to assign it a controller.

Let's create that first title controller is equal to text editing controller.

Assign the title controller.

And then I'm going to start writing on its decoration input decoration and assign it a label text which will be or title.

The first input we're going to be taking from the user.

Next let's save this class.

And I'm going to go to my home and on my floating action button I'm going to add the navigation.

So navigate to George Bush and assign it my page transition.

I'm going to set the type to page transition type dot page.

And then I'm going to pass it the screen I want.

So the child will be create poll screen save this also okay so let's run our program.

So let's try clicking on our button.

And it's taking us to the Create Ball screen.

And we have our first text field.

So now to further enhance our UI I'm going to wrap my text form field With a container widget.

And I'm going to apply my decorations on that.

First I'm going to set its margin property.

So the margin is, you know, the inner padding from the content.

So it's going to be a little padded from the inner text form field.

Next I'm going to set the decoration and box decoration I'm going to set the color to colors or white

so it blends in with my background.

I'm going to set the border radius to radius dot circular and set to five.

Next I'm going to do the border border dot all.

I'm going to set the color to colors gray dot with opacity.

So to make it a bit translucent.

Next I'm going to add a box shadow and I'm going to set its color Other two colors George Grey dirt with opacity and 0.2, and set the blur radius to six.

How far we want the shadow to spread and spread radius will be one, and finally offset will be offset or zero and three.

So I'm just going to wrap this and that's it.

Let's check out our text form field.

Now it looks much better as you can see.

So now what we want to do is create three options.

So for that I'm going to create a for loop in my UI outside of my container.

So I'm going to be giving the user a maximum of three options to select from.

So I'll say I is less than three.

And then I plus plus.

And I'm going to create a container and I'm going to give you the same styling as above.

So I'm just going to paste this here.

Next I'm going to create a child for the container.

First I'll create a row and then I will be passing it a text form field so that the controller we need to assign multiple dynamically.

So what we're going to do is create a list of controllers.

So list of text editing controller.

I'm going to call it option name controllers is equal to.

And I'm going to set it to three text editing controllers.

So let's copy this and paste it two more times.

And I'm going to go back to my list of controllers.

And here I'm going to pass it option controllers name of I.

So whichever index you want pass that controller to it.

And then I'm going to set its decoration.

So increased decoration and set the label text to option and wrap it for interpolation.

I plus one because it's going to start from zero and just name.

We'll wrap our text editing in a expanded widget.

So let's save this and restart it.

Let's go to our Create full screen.

As you can see our title field.

And then we have our three options which we've rendered dynamically.

To summarize what we've done today, we've structured the UI to ensure responsiveness across different screen sizes by using text form fields.

We made it easy to manage user input dynamically.

Finally, we've applied smooth transition using navigator and page transition.

We created a seamless user experience by carefully using margins, paddings and checklists.

That's all for this video.

Thank you for watching and I'll see you in the next one.

3. Create Poll Screen - Part II

Hello everyone!

Today's video is a continuation of the previous one.

In the last video we built a pole screen.

Now we are going to continue by working with image selection in our app.

First we will add some spacing and ensure images are displayed correctly.

Then we will integrate the image picker to allow users to select images from the gallery.

Finally, we'll handle the selected images efficiently to ensure a smooth user experience.

Now that we understand the key steps, let's move on to its implementation.

Okay, so to start working with our images, I'm going to first create a list of images which I can store it in.

So option images now and we're going to need exactly three.

So let's set them to null first.

And set its type to list of file.

And then let's start working on images.

Under my expanded widget, I'm going to create a size box to add some spacing.

Weight of 15.

And then we have to make it optional that if we've selected an image there we want to show the actual

image.

And if it's not been selected yet we just want to show an empty container.

So if that index of image is not equal to null, then what we want to do is show the image.

So image dot file and the path will be stored inside our options image of ith index.

And it will not be null because we're checking before.

And then I'm going to set the height and width to 50.

This basically tells us that since we've set our file image to question mark, which might make it null.

So if we add this, it means that we are certain at this point the images will not be null because we're checking them here first.

Next, I'm going to set the fit property to box which dot cover and if image is null, we just want to show a container.

So I'm going to set its height and width to 50.

And then I'm going to set the color to colors dot gray 500 a bit less.

Let's make it 300 instead.

And then I'm going to add also a child.

And I can I can start image to show that the image has not been selected yet.

And when we're showing our image I'm going to wrap it in another widget called the clip rect.

So the image, whatever shape it's in, we want to show it in a rectangle.

It's specifically our rigged and set the border radius to border radius dot circular and set it to ten.

So let's save it and check it out.

And as you can see, we can see our containers for now because we have not selected any images as of yet.

Also let's add a padding here along with margin padding.

Edge inserts to it all and then save it.

Now it looks much better.

Let's add decoration to our container and box decoration.

I'm going to set its border radius to border radius dot circular and set it to ten and move my color inside of my box decoration.

Because as we know, it's going to throw an exception if we've declared them separately.

So let's save this.

Check it out.

Now you can see it's rounded from the corners.

Now, what I want to do is when user clicks on our containers, we want to open an image picker, which basically lets access to the gallery so the user can pick images.

So for that I'm going to import another dependency called the image picker and import that.

Next I'm going to go top of my file and I'm going to create another variable image picker.

And I'm going to name it Image Picker and assign it an instance of image picker an object.

Now I'm going to go to the bottom and I'm going to create a method for picking images.

So make it a feature method of void.

And I'm going to call it pick image and just pass it an index.

So final picked file is equal to A weighted.

Image picker George Pick.

Image method and image source is going to be gallery because you want to pick from gallery.

You can also choose to pick from camera.

And then I'm going to see if the picked file is not equal to null.

If so I'm going to assign it to my option images of I th index because we passed that and I'm going to assign it picked file dot path.

You don't assign it the actual image just assign it the path set stated.

And then what I'm going to do is go to my images part and create a gesture detector widget.

And to each child I'm going to assign this images.

So copy this and drop it.

Now on its own tap function when the user taps on this, I'm going to need it to call the peak image function and pass it the index.

And that's it.

So let's reload it and check it out.

Go to our poll creation screen and tap on the container.

And image Picker has opened.

So select any image and it's going to start showing here.

Select another one.

And as you can see if our image is not selected it just shows a container.

And if it's selected it shows the image.

Now if you want to select another image you can just click on the container again.

It's going to open the image picker.

That's it.

That's how you use image picker in your flutter apps.

To summarize what we've done today, we have successfully implemented image selection in our flutter app.

Now, images display properly and users can pick images seamlessly from the gallery.

This process is smooth, efficient and enhances the user experience.

Thank you for watching.

I hope you found this video helpful and I'll see you in the next one.

4. Create Poll Screen - Part III

Hello everyone and welcome back.

In the last part, we built a ball screen and added image selection in our app.

Now we're continuing with poll submission feature.

First, we add a button that allows users to submit their polls easily.

Then we'll ensure smooth input handling, making it simple for users to enter their responses.

Next, we'll add validation to prevent errors and keep data accurate.

Finally, we'll securely store the submitted polls in Firestore and ensure a seamless user experience.

So now that we know what we're going to be doing, let's get started.

So I'm going to start by creating an elevated button.

Let's add a height first and then our elevated button.

Set it to nothing for now.

And to the child I'm going to assign text that is create ball and I'm going to style it.

Textile on size will be 14 and the color is going to be colors dot white.

Then I'm going to set the decoration for my elevated button.

So elevated button dot style form.

And I'm going to set the background color to color stored orange accents.

And I'm going to share the shape to rounded rectangle border and border radius.

I'm going to set to border radius to circular and eight.

Next I'm going to set the padding property.

And as you said symmetric horizontal is going to be 16 and vertical is going to be ten.

Let's save this and go to our pro screen and you can see our create folded.

So now I'm going to go to my home show full screen class.

The first thing I'm going to do is get my current user ID.

Current user ID.

And I'm going to use Firebase auth dot instance and dot current user.

That gives us our current user's ID.

Now I want to pass this ID to my current create bulk screencast.

So for that I'm going to have to create a parameter here of final string current user ID.

And then I'm going to create it here as required.

So it's always needed whenever this class is called.

And now I just pass it to where it's calls.

So current user ID is current user ID.

Save this.

So next I'm gonna need to do is make my home screen go from login screen.

So we always have the auth ID.

So I'm going to connect it to my sign in screen.

And here once sign in is complete what I want to do is go to my handle sign function.

And here in handle sign in I'm going to check and match the string.

Let's paste this navigation code first.

And I'm going to check if the string is sign in or successful message is equal.

Equal to sign in.

Sign in was successful.

In that case.

What I want you to do is navigate to my home screen.

So import this.

And instead of this I'm going to say home show screen save this and rerun our app.

Now it's going to navigate to our sign in screen.

Let's log in.

Sign in.

And as you can see now it's taking us to our home screen.

So next in my screen I'm going to be creating a new method for checking if my form is valid for submission

such that all the fields are filled up, the option and title fields.

So first thing I'm going to pass it is the build context context.

Then I'm going to be checking if title controller dot text field is empty.

If it is empty, just show a scaffold messenger dot of the context and show snackbar and pass it a sidebar.

I'm going to set its text to title is required.

Next I'm going to be checking all my options field.

So for that I'm going to need a for loop.

So I is equal to zero I is less than option name controllers dot length and I plus this.

If my option names controller of I th index dot text is empty, I'm going to show another snackbar.

Copy this and replace the text option of I.

Plus one name is required and from here return false.

Also return false from here.

And then next to the text I'm going to be checking if option images dot contains now.

Then show another scaffold and say, just generically, please select an image or all options and return false from here also.

So false otherwise if it all works and there's nothing, return false, just return true.

Then I'm going to start with another function which will be my submit poll function submit poll.

And we're going to be passing it the user ID and the build context.

So first I'm going to here validate my form.

And just return.

If it's not valid add context here.

Then I'm going to go to my on press.

And here I'm going to call my submit poll functions.

I'm still going to add the widget dot current user id and context.

Next I'm going to be adding a couple of dependencies.

The first one is going to be Firebase Storage for storing my images in Firebase.

And then I'm also going to be adding another dependency which will be Cloud Firestore.

So now we're going to start working with Firebase for storing our polls.

Let's remove this semicolon.

And here I'm going to start working inside a try catch block.

First I'm going to get a poll doc.

So for that you have to call Firebase Firestore dot instance dot collection.

Collection is basically a table in Firebase.

So polls Dutch doc.

So what this is going to do is create a new document in my collection and then I'm going to get its

ID.

So call ID is going to be call doc Dot I.

Next I want a list of map of string dynamic.

And I'm going to call it the options list.

So next I'm going to start a for loop zero I is less than three and I less than.

And then I'm going to check the image URL and pass it a function that will return an image.

So I'm going to pass it the image option images of I and the index.

So let's go ahead and create it.

So another thing I need to add to our upload function First is the id.

We need to pass it so we can add it to Firebase.

So call it also and make this notes now.

So let's go to our function and remove this.

Next I'm going to add a string of all ID.

And let's start working on our function I'm going to make it return a string for now and make it nullable.

So I'm going to wrap it in a try catch block first.

So final storage reference of Firebase.

So Firebase storage dot instance dot ref.

George I'm going to create the path now.

So Paul and then I'm going to concatenate my poll IG.

And then I'm going to concatenate the index.

So either two string and again reference it with all ID dot jpg format.

And that's it.

That's the name we're going to be setting for our poll.

Since the poll ID will be unique every time or each poll, we don't need to worry about images being overwritten.

So I'm going to hit the upload and I'm going to assign it.

Let's async a storage ref dot put file.

This is basically going to upload my file and option image.

Let me rename this to image.

And then I'm going to just after it's done uploading return it.

So upload image.

Church Rev George get download URL.

So basically the parts so you can access the image.

And in our case I'm going to first log it so we can debug it if there are any issues.

And I'm going to say error uploading image.

Just concatenate the error and I'm just going to say return null in this case okay.

So now we can go back to our function call.

And here you will see it's very important to handle errors because if we can check here if image URL is null that means image was not uploaded correctly.

So in that case what we're going to do is throw an exception.

Image upload failed and I'm going to await this also So.

So you can wait for the results.

Now that we've checked that our image will not be null.

What I'm going to do is options.

So let's add a map.

And the first value will be its name.

I'm going to assign it option name controllers of I and its text.

Then the second value will be the image URL.

And I'm going to assign that image URL directly which we returned.

And finally we're going to set the value to zero which will be how many votes it has been assigned, how many times it has been voted on.

So for now, when we've created the poll, it's not going to have any votes.

Now I'm going to go ahead and write my document to Firebase.

So I'm going to say poll doc which is my reference.

And I'm going to say set its value.

And to that I'm going to assign poll Paul I'd.

Set it to Paul.

I'd.

And another thing I want to create in this is title.

Set it to title controller dot text.

Can we have our options field which we've created.

So I'm going to call it options and assign it my option.

Then I'm going to create a created edge field which will be a timestamp for when it was created.

So I'm going to set it to field value dot server timestamp.

Then I'm going to create a creator ID field.

And I'm going to assign it the user id.

And finally we're going to need a total works field to which we're going to assign zero because freshly it won't have any words.

So it is also important because that's how we'll know which user created the polls so we can show their data to the user.

Now that we've created our submit functions let's go to Firebase.

And here I'm going to go to build.

And first I'm going to go to Firestore Database.

And here you're going to check the rules and set them to true.

Because by default they're false.

And they're not going to allow you to write any data in your database.

And secondly in your storage you're going to be storing your images.

You also want to set the rules to true because these are also going to be false by default.

So make sure both are true so you don't get any errors.

And then finally I'm just going to go back to my code.

And here I'm going to show a message saying that my poll was created successfully.

So for that I'm going to copy this and paste it right outside my loop.

So here and replace this with poll Created successfully.

And then what we want to do is go back to our home screen.

So for that I'm going to do Navigator George.

Bob, I see the context.

To create a loading state I'm going to first create a loader.

So final loader is equal to false by default.

And go back to my function submit poll.

So right below my validation if it's true I'm going to search state it and set the loader to true.

Let's make a boolean and go back.

I'm going to copy this.

And then finally right below my catch when either if there was an error or not, I want the loader to be set to false.

So I'm going to set it in the finally section, and I'm going to create a snack bar in my error state.

Replace the text with failed to create call.

And I'm also going to remove this navigator, because what this is going to do is stop my scaffold for showing, because the section is going to show, the neckline is going to execute, and it's just going to cause the navigator and close my scaffold again.

So I'm going to remove that and let's save this and run our app.

So go to create full screen.

And I'm going to create a poll.

Which color do you like.

And I'm going to give it a couple of options which is yellow orange.

Or purple.

And I'm going to select images.

Just select any images from your gallery.

And a third one.

So let's create our poll.

And as you can see our poll was created successfully.

So let's go to Firebase and see if it was added to our collection.

So Firestore database.

And in our collection of polls we've got our data that was added on this stage with our options.

So that's it.

That's how we store our data in database.

So we've successfully implemented the poll submission feature.

We added a button that allows users to submit their polls effortlessly to ensure smooth input handling.

We implemented validation, preventing errors and keeping data accurate.

Finally, we securely stored the poll data in Firestore, making it accessible for future use.

By focusing on these key steps, we've improved the user experience and data integrity.

Thank you for watching and I'll see you in the next video.

5. Updating Image Upload Logic

Hello everyone!

Today's topic is fixing an issue that was in our Create poll screen.

We'll fix an issue that happens when creating a poll in our app.

Right now there is a problem with image validation which users can proceed without selecting an image.

We'll go step by step to add proper validation, update the logic and make sure the poll creation process

runs smoothly.

Finally, it will test everything to confirm that the issue is resolved.

Let's get started.

So in my validate form, it's not properly checking if the user has uploaded an image.

For that, I'm going to apply an Ldif to check both condition if one is true.

If not, then check the other one.

Not unconditionally checkable.

Next I have my upload image.

Here.

I will replace this.

I think I added underscore something here, which is supposed to be a slash that tells us this needs to be a folder also.

So we're just going to do is create a folder and then a folder with the ID.

And then it's going to store the images in there.

Next, let's check out our.

Submit again and see if there are any issues here.

This one appears to be fine.

And finally I'm going to check my submit button.

And here I forgot to add my conditions my order logic.

So I'm going to add that here.

So loader this enter widget.

And I'm going to assign each child a circular progress indicator.

Otherwise show the elevated button.

And I'm going to set its color to colors dot orange.

And let's say let's rerun our app.

Let's login.

Sign in.

Okay.

So I'm going to create a new poll.

Now let's create a question.

What foods do you like.

Like the most.

And let's say Burger.

Or fries?

And let's try submitting the poll without selecting any images.

And as you can see, we get disclaimer for having to select the images first.

So select first one.

Let's try again.

Now it's going to wait now for all images to be selected before trying create a new poll.

And second one.

And finally the last one.

Let's try creating a poll now.

And as you can see our loader is appearing and our poll was created successfully.

We've successfully faced the issue by adding proper validation for image selection and improving the way images are handled.

Next, we define image upload logic to ensure images are properly stored and retrieved, preventing potential errors.

After testing, we confirm that everything works smoothly.

Now user won't face errors when creating a poll.

Thank you for watching this video and I'll see you in the next one.

6. Update All & Posted Tabs

Hello everyone!

In this video we'll be working on our home screen to show our latest posts.

First, we'll create our poll Card UI, in which we'll show our poll data such as the image and its options, along with the title that will show all polls that were created by the current user.

And then in the All Posts tab section, we're going to show all the posts that were created by other users also so the user can watch on them.

So now that we know what we'll be working on, let's get started.

So I'm going to start working on my home screen.

And for that first I'm going to create a new widget.

I'm going to call it Build Poll Card.

And to this we're going to be passing a document snapshot, which is basically a Firebase reference that tells us that we're referencing this particular document at the moment, so we can be referencing

any one document at a time.

So this will give us access to a single document and I'm going to extract its data.

So I'm going to do is folder data.

Give me the data inside this as a map of string and dynamic something I can consume in flutter.

And then I'm going to create an options to extract the options from my data.

So I'm going to say data and its options as a list of dynamic.

So it's going to get me the options in this options variable.

Now I'm going to start working on its UI.

So return a padding widget because we want to add some padding first of all.

So edge inserts short symmetric.

And I'm going to set the horizontal to 20 and the vertical to ten and the semicolon.

Next I'm going to add a child.

And it's going to be a container because I can easily design that.

First let's create the decoration box decoration.

And I'm going to assign it the color color stored white.

Next I'm going to assign it the border radius and border radius.

So it's circular and set it to 15.

Sharp edges tend to look bad so always try to add borders radius.

And next I'm going to add a box shadow.

So box shadow set its color to color.

Search gray George with opacity I'm going to set it to 0.3 to make it empty, translucent, and then

I'm going to add the blur radius to blur it and the spread radius to two.

Not that far from the card, and I'm going to offset it to offset zero.

And then on the right side I want it to be five.

Then we're going to create the child of our container.

And I'm going to assign it padding.

And agents that should all and I'm going to set it to 15.

Then I'm going to create a child.

I'm going to assign it a column because we want a lot of data in it.

I'm first going to set its cross axis alignment to start because by default it's set.

And I want data to appear in the start and replace this with children property.

Now I'm going to start working on it.

First I'm going to create a row.

And I'm going to set its main axis alignment to main axis alignment or space between.

I want space between the children.

And I'm going to first create an expanded widget in it, assign it text, and the text is going to be my data of title.

So it's going to fetch the title property for my data.

And I'm going to set its text align to text align.

Center.

Then I'm going to start styling my text.

And for that I'm going to use a textile property.

And I'm going to set the font size to 18 and font weight to font weightbold.

And finally I'm going to set the colors to not exactly black.

I want it to be a bit color, so I'm going to set it to black 87.

So what I'm going to do next is our side, my row.

I'm going to create a divider to add a surf line between my title and my options.

I'm going to set the line color to orange accents and set its thickness to one and its height to 20.

Then after my divider, I'm going to create a size box with a height of ten.

And then I'm going to start working on my options.

For that I'm using a for loop so I equal to zero I is less than options dots length.

And I plus plus.

Now what it's going to do is create this children's list.

First thing is going to have is my container.

I'm going to set its margin to edge insets.

And I'm going to set the padding to.

store it all.

And I'm going to set its value to 12.

Next I'm going to work on the decoration box decoration and set the color to color Dodge white.

Border radius is going to be border radius.

So it's circular.

And assign it 12.

Then I'm going to have my border property.

And I'm going to say show all sides of the border.

And I'm going to set its color to colors to gray to pasty and show it with 0.2.

Finally we have our box shadow.

And I'm going to assign it a shadow.

And I'm going to set it colors to colors dot gray dodge with opacity again of 0.2.

Also.

Then I'm going to set the blur radius.

Days two six and the spread radius to one.

And the offset is going to be offsets of zero and three.

Now I'm going to assign the child a row widget.

And I'm going to set its cross axis alignment to cross axis alignment to center.

Next I'm going to use a clip art rect widget.

And I'm going to set it border radius to border radius.

So it's circular.

And I'm going to use this for showing my image.

So I'm going to say image dot network reuse the network.

When we use showing an image from a URL.

And the asset when we're showing an image from our folder of assets.

So here I'm going to pass it my options of I and the image URL L property, and I want to set its height to 50 and its width to 50.

Also, by default, no matter what size it is.

And I'm going to set its field to box fit dot cover.

And if there is an error while getting my image from the network, I want to use the error builder.

So I'm going to pass it the context, whatever the error is.

And this stacktrace.

And if there is an error in that case, we just go ahead and show an icon widget.

Set the size to 50.

And then after my break I'm going to add a size Each box of width 15, and the next video I'm going to add is an expanded widget.

And I'm going to assign it a column widget.

Set its cross axis alignment to cross axis alignment, or start and set the children to text.

And the text is going to be option of I and then name property.

Then I'm going to style this.

Also, let's just copy the text for title and its style.

And paste this here.

We don't need to align text to center here.

So I'm going to remove this.

Set the font size for name to 16.

The name is basically going to be the option value.

Whatever options we provided and 87 is fine.

Then I'm going to show a size box.

Of height eight.

And after that I'm going to show a linear progress bar for showing how many times it's been voted on.

So I'm going to set its value to options of I and the words property.

And then if it's null, set it to zero and divide it by ten.

And it's going to show progress according to that.

So colors dot orange.

And I'm going to set the background color to colors church gray.

And finally I'm going to create the vote button to allow users to vote.

So I'm going to say size box and set its width to 15.

And an elevated button.

Set its own breast to nothing for now and replace the child with a text widget.

Let's call it word and I'm going to set its style property.

Remove this, set the font size to 14, and I'm going to remove the boldness.

I want regular text and I'm going to set its color to white.

And then I'm going to style my elevated button.

I'm going to use a style property.

Dot style from set the background color property to colors dot orange accent.

In shape.

Two.

Rounded rectangle.

Border.

Border radius and border radius dot circular to eight.

Use its padding so edge inserts dosimetric.

Check the horizontal to 16 and the vertical to ten.

Okay, we can't directly view this.

Willpower card so we have to create our widget.

Underscore build all posts widget.

And it's going to return something special.

It will be a string provider.

And basically it listens to a stream.

And if there are any changes in Firebase collection it's going to reflect that immediately.

For example, a new user puts a new poll.

It will be shown immediately in our all polls section.

So first I'm going to replace this stream.

Add a query snapshot document type here.

And I'm going to start replacing the stream with Firebase Firestore and its instant start collection of polls.

Dot pair creator ID.

We don't want to show our own polls.

We want to show other people's polls in our posterior section.

So I'm going to show is not equal to underscore current user ID.

And then I'm going to get dot snapshots.

Then I'm going to be replacing my builder with context And index parameters and snapshot.

Then I'm going to check if my snapshot data dot has data variable, and if it's true, make it false and return a center widget.

That will be a child of circular progress indicator.

Which color is going to be colors dot orange.

And if it's not the case, I'm going to get my polls is equal to snapshot dot data, which will not be empty at this point because we've already checked and I'm going to get all the documents of Firebase that were in that collection.

So I'm going to do is return false.

Dot is empty.

If it is still empty I'm going to show.

center child and a text that will say no ball available and I'm going to style it.

Text and I'm going to set the color to colors dot orange.

Font size is going to be 20.

And finally font weight is going to be font weight 500.

And here I'm going to show a list view builder.

Builder.

And I'm going to be passing it my context and index I'm going to set its item count first which will be false dot length.

Since we've done most of our UI building logic, we're not going to have to do a lot here.

So I'm going to say false of Fix.

Just get the singular pole at a time and return our build pole card.

And that's it.

You just have to pass it the pole and it will show us our pole.

Let's copy this.

And I'm going to paste it here so we can show our poles here.

So let's restart this.

And as you can see our all section is showing our poles that were posted by another user.

I'm using one account here.

And if I log in using the other accounts through which I posted the poles, it's not going to show anything in the all tabs section.

Next I'm going to work on the pole section.

So for that I'm going to create a new widget.

Visit Wales for states polls.

He's also going to be running a stringbuilder, so let's just copy this and recreate it here.

I'm just going to rename it build post it false.

And instead of not equal to I'm going to post is equal to.

So it's going to show all the posts that were posted by this user.

So I'm going to replace the text here to posted poll available.

So I'm going to copy this and replace it here.

Let's try reload it.

And now if I'm in my posts section as you can see I can see the polls that I have posted and nothing in the old section.

I'm going.

I'm able to see the image and if users have posted on this since we have implemented post rotate functionality,

this progress bar is empty and we can see our option titles.

Similarly, if I log in using another account where the user has not posted any posts.

Sign in here you will see all the posts that were posted by that other user, but in the posts section,

you're not going to see anything because none of the posts have published by this user.

So in this video, we updated our home screen to show dynamic data.

We've updated the polls tab to show polls created by the current user, and then we also updated the

All polls tab to show all the polls that were created by other users.

So that's it for this video.

Thank you for watching and I'll see you in the next one.

7. Edit Poll Screen

Hello everyone!

In today's video we will look into the edit screen with Firebase.

We'll dive into how to add edit functionality to our app.

We'll start by setting up the UI for editing polls.

Next we will add, edit and delete buttons with functionality.

Then we'll pass data between screens for editing.

Lastly, to implement the logic to update poll details like title option and images in Firebase.

By the end, you'll know how to make changes in the data stored in Firebase.

Let's get started.

So to start I'm going to need edit and delete buttons.

So let's scroll down to the end here.

I'm going to create a size box first.

With height of five.

Then I'm going to create a row of buttons.

I'm going to set its main axis alignment to main axis alignment to it and then the children property.

I'm going to add a text button here.

Text button dot icon on press is going to be nothing.

And I'm going to replace the label with text edit.

And I'm going to set its color to orange.

So color colors to orange.

And next I'm going to also add an icon for it.

So I can.

I can start edit.

And I'm also going to set its color to orange.

Next I'm going to create a size box with a width of five.

And I'm going to copy this button and use it for my delete.

Also, just change this color to red.

Here also.

And I'm going to change the text to delete.

And I'm going to change the icon to delete.

Next I'm going to navigate this to my edit screen.

So I'm going to say navigator dot push.

And I'm going to replace the route with material page route.

This is going to be the content.

Edit all screen.

And I'm going to set a property in it to which we assign the port.

I'm going to name it edit call screen dot dot.

Make it a stateful widget.

visit.

And here we're going to create a parameter to final document snapshot.

And that's what we're dealing with here.

I'm going to call it poll.

I'm going to make this a required parameter.

And this torch poll.

So let's import this here and save.

So my default class is going to be almost the same as my create poll class.

So I'm going to copy.

These variables from here and add them to my edit screen.

Then I'm going to import these files.

And with a list of options I'm also going to need a list of string.

I'm going to call it existing images because it's going to be a part of my snapshot already.

Now in our Since.

We also have another important method which we override from our parent classes, we call it the init

method.

Now what this does is whenever the class edit post screen is created, the first thing that's going to happen is that init state method will be called.

So in here you can perform any actions you want.

As soon as the class is created.

So for example I want to extract my data from my poll which I sent as a parameter here.

So I'm going to access it with widget dot ball dot data as a map of string and dynamic.

So the first thing that will be happening when this class is created is the data will be extracted into my data variable.

Next what I want to do is to my controller I want to assign data title property then to my options.

I'm going to give it a map of string and dynamic dot from so directly.

Data.

Options.

It's just like we're setting it as map of string dynamic here converting it.

We also want whatever data is to be a list of map of string and dynamic.

And we're then assigning it to our options.

So for option or option in options we have our list of controllers.

And we want to assign a data.

So option control is stored as text editing controller because we already have the text.

We don't want them to be null and assign the text.

So option of name variable and also existing images dot add.

I want to set it as option of image URL.

Make sure that you're not making any typos in this because data will not be transferred properly otherwise.

Also, when you're creating any you have to also super dot init state.

Next let's start working on our scaffold.

So scaffold first the app bar.

I'm going to set its title to a text of edit poll.

And then I'm going to say center title is true.

Then for the body I'm going to assign it a single child.

Scroll view first and to the child a column of children.

So now to instead of creating a text view from scratch again, I'm going to copy it from here.

To save time.

And next I'm going to copy the image logic also.

So I'll copy this.

Up to here.

Add this below my container.

For now I'm going to remove this method from here.

Next I'm going to check here if my existing images of I is not equal to null.

Then we can show this.

I'm going to make this available.

At this point, it won't be there because we've assigned it an empty array.

Next I will replace this with images dot network.

And this will go here.

And one more thing we need to do is remove these because we don't want to set them to anything because

we're adding fresh ones here.

So let's restart this.

Let's enter our email and password.

Sign up.

So in our pursuit section we can see our edit and delete button.

Let's click on edit.

And as you can see we are doing our question and our text and images preloaded.

Next we want to show an edit button here.

So once this is done editing images or text or title they can click on edit and the data will change

in Firebase.

So for that I'm going to copy the button from here.

And I'm going to paste it at the end.

I'm going to remove this function and name this into edit or let's save this.

And now you can see your edit button.

Next I want to change the functionality so user can pick new images here.

Let's go to create Post screen and copy this from here.

And add it.

Next I'm going to call it inside my gesture detector right here.

So pick image.

Now what this will do is pass in the index.

So it will only be replacing that image.

So let's say this.

Ages.

And want to change this image for example.

With this one, as you can see, all the other images are same, only this one has changed.

What's happening here is it's checking if option images are null, because when we get to this screen they are going to be narrow, right?

Because the thing is going to be signed and it's going to check if existing images are null and existing images are going to be coming from our snapshot.

If user has not posted any images by any chance, or Firebase is not providing it at the moment, it might be null.

In that case this container will show otherwise our existing image will show.

Next I want to create the update function.

So I'll copy this validate form function from here.

And paste it below my image.

And I'm going to remove this image from here, because I'm only going to be checking if the title and options are correct, because images will already be populated in our code, because what we'll be doing is if you pick the new images and if there aren't any, it will just use the previous one so the images won't possibly be null.

Next, I'm going to copy my submit poll function from here.

And rename it to.

Update poll.

And instead of user id I'm going to be passing it the poll ID.

And from here I'm going to remove the warts because I've already users have voted.

It will be overwritten if I set it to zero here and in my folder.

Besides title and option, I'm going to be removing everything else.

Next I'm going to be removing this from here.

And I'm going to.

I'm going to create new variables here.

If image URL is equal to existing image url of, I assign this.

And I'm going to check if option images of I, which basically means if the user has updated this image is not equal to null, then I want the image url to be the uploaded image URL.

So let's copy the upload function from here.

Copy this and paste it here.

Import it.

And here I'm going to say await.

Upload image I'm going to rebrand it to shrink and rename this image.

Let's add this to option images of eye.

Let's import this lobe library developer.

And then inside my edit pool I'm going to call my update function.

And to the pole id I'm going to pass widget.4.id.

So let's save this also in my update function.

Since we are updating the pole node adding it, I'm going to be removing this line because we're not creating a new document and remove this also.

Remove this answer.

And now outside my for loop I'm going to use the update function instead of the set function.

So I'm going to say await Firebase Firestore dot instance dot collection of posts dot doc.

Because we have the doc reference which is the ID and I'm going to update it basically here the ID is also the doc ID so we can just access it using that.

I'm going to make it a map here and set the title to title controller to text, and then another properties in option which will be option.

And I'm going to change the text to edit it successfully able to edit poll.

And another thing we need to do here is set our data to null or empty.

So it's not already populated with anything.

So I'm going to set it to.

Clear.

Copy this.

And for my option images also set it to clear.

And finally my existing images have been set to clear.

Also it will not be null here.

And here I'm going to add three nulls to my option images dot add.

Now let's recheck if Harvard this time before submitting seems to be fine.

So let's restart it.

Enter image and password.

Sign in.

In my post section I'm going to edit this file for which color you like.

I'm going to set it to one Two and three and most.

I'm going to take a screenshot and replace my first image with this one and try editing by Paul now.

And our Paul was edited successfully.

Let's go back.

And as you can see image is updated.

And along with the option text and our title text.

So let's see that's how you change or update your data in Firebase.

Here's what we accomplished today.

We paired the edit screen with clean and functional UI.

We added edit and delete buttons with all the necessary logic and pass data between screens to make editing seamless.

We updated all details like the title option and images in the database.

Thank you for watching and I'll see you in the next video.

8. Delete Poll from Firebase

Hello everyone!

Today's topic is deleting data from Firebase.

In this video I will show you how to implement a delete function for polls.

We start by adding functionality to the delete button.

Then we're going to handle the poll data and any associated images stored in Firebase.

I walk you through deleting both the poll document and its images from Firebase Storage.

Finally, we'll add user friendly messages to confirm the deletion or handle any errors.

By the end of this tutorial, you will be able to delete data seamlessly, including any images from Firebase.

So let's dive right in.

Now we're going to be handling the delete functionality here.

We don't actually need to create a separate screen for this since we're only deleting the polls.

So for that I'm just going to go in my home screen.

And I'm just going to create a function right here.

So future of.

Word and I'm going to call it delete ball.

I'm going to be passing it to parameters.

One will be the ball ID and next will be the build concept.

I'm going to wrap it in a try catch block.

Document snapshot.

I'm going to get it fresh.

From Paul Joe fetch the document from Firebase Firestore instance dot collection of Paul's dot document.

Dot get.

And then if the ball doc dot exists I'm going to start working.

I'm going to first create a list of dynamic and options.

I'm going to name it I'm going to fetch pitch the option for my ball jock.

Options.

Otherwise, assign it an empty list.

Next, I'm going to create a for loop option and options list.

Option of image URL is not equal to null.

I'm going to fetch the URL option.

Of image url.

Then I'm going to get this file path.

Make this nullable.

Then I'm going to get its file path Uri George decode for uri George.

Pass image url dot path segments Dot.

Square element.

Element.

Dot contains ball or else go get null since we saved our image as dot underscore id.

So what is going to do is check where we have the name hole and it's going to just fetch the image.

Otherwise it's going to recursively go through the full folder path that we have fetched for my image URL.

Next I'm going to check if my file path is not equal to null and my file path is also not empty.

I'm going to say await Firebase storage dot instance dot ref file path dot delete.

And that's how we will be deleting our image from our Firebase Storage.

And once it's recursively gone through all of this and deleted all the related images for the three options we have, I'm going to step out of my loop and I'm going to delete my whole document.

So Firebase Firestore instance dot collection of posts George document.

Give me the reference and just call the delete.

Then I'm going to show a message here.

So scaffolds messenger dot all.

Don't show snack bar and create snack bar widget.

In this content I'm going to pass it a text widget and the data will be.

All and associated images.

Deleted successfully.

Let's copy this and paste it in a catch block.

Here I'm just going to show my error message.

So error deleting call.

And this concatenate my hair.

And in the delete here.

I'm going to call my delete function.

Replace it with Paul George ID and add a semicolon.

Let's save this.

So now we have our two poles I'm going to delete the second one.

Which color do you like.

We have our pearl and then our documents.

This is the one I want to delete.

And it's pole id is w BQ.

So I'm going to go into my storage.

And in my poles folder you can see that the data is stored in my ID.

So let's go ahead and delete it.

And it says Paul was deleted successfully.

Let's go back.

And as you can see now when I go into my Paul's folder, a folder with w BQ is not existing.

And in my Firestore database there's only one document left.

The other one has been deleted successfully, which successfully implemented the delete functionality for our posts.

We first fetch the Paul from Firebase, check if it had images, deleted those images from storage, and finally remove the Paul document.

To make our app reliable, we wrap the delete function in a try catch block to handle issues.

After deleting a Paul Fisher confirmation message using Snackbar.

This way, users get feedback that their action was successful, improving the overall experience.

Thank you for watching this video and I'll see you in the next one.

9. Handle Votes & Show Voters Tab

Hello everyone.

We're going to add more functionality to our polls.

Here's what we'll cover today.

First, we'll set up a logic to handle polls.

Then we'll update the database to store user words.

We'll also make sure users can vote only once.

Finally, we'll display the polls.

Each user has voted on making the experience personalized.

So let's dive right in.

So I'm going to start by creating a word function.

So future of.

Void.

And I'm going to call it handle vote.

It's going to have a couple of parameters.

The first is going to be a poll ID and then.

An option index build context context.

And then finally it's also going to have a current user ID.

Let's make it async.

I'm going to wrap it in a try catch block.

Now I'm going to create a variable that's going to fetch my reference for Firebase document.

Dot doc.

And then I'm going to create a new variable called snapshot.

And I'm going to pass it all doc and get the reference.

I'm going to check if holder dot exists.

Then I'm going to fetch the data from it.

Don't data as a map of string.

And dynamic.

Then I'm going to create a total variable and I'm going to retrieve the words from my data.

So total quotes.

And finally I'm going to get my options from data options as a list of dynamic.

Doesn't matter if they are integers or strings.

So I'm going to create a loop to loop through my options.

If my option of water's.

Not equal to null.

And ocean of Waters.

As a list.

Contains current user ID.

Then show a message.

So I'm going to copy this paste it here.

I'm going to replace this with the text you have already watched.

So basically what this is doing is checking if waters are not null and if the waters list contains the current user ID the current user.

That is what.

So if the user has already worked, we don't want to allow him to vote again.

And after checking all this, if it's true, I'm going to return it.

Next I'm going to get the water from Fun options.

Option index.

And the waters from here.

Otherwise, return an empty list.

I'm going to do it.

Add current user.

Right.

So now we're going to add our users vote.

And we're going to also add it to the list.

So the user cannot vote next time.

And it's waters is equal to waters.

Then I'm going to say options or option index.

And the votes property is equal to.

Options index.

And the words property.

If it's zero and we just add our new board in it.

Next what I'm going to do is update it in Firebase.

So for doc dot update options.

As options.

And next I'm going to do call doc dot update.

Total votes.

With total votes plus one.

Then I'm going to show another messenger here and here.

Replace this text with word submitted.

Successfully.

Fail to submit.

And that's it.

Now, what we're going to do is go to our work section of our home screen.

And in the on breast, I'm going to call my handle word function and replace the ID with bold or ID.

And option index is going to be I.

And replace the current user id.

Next I'm going to go to my build poll function build poll card here.

And I'm going to make it optional.

Then the user pass a user.

What variable?

Assign it false by default.

Now inside my function I'm going to go to my vote button and I want to show the word vision only if the it's not the current user.

We don't want the actual user to be voting on the poll themselves, and on our edit and delete buttons.

We only want to show them if the poll was also posted by the same user.

So not of show user.

The actual creator of the poll can make changes.

So we're saying not show user.

So in our linear progress indicator we also want to show if user has voted or not.

So first we're going to create a condition show user vote.

And.

The same condition we created below Options of Eye of Waters.

Is not equal to null and.

Options of waters as a list.

It contains current user ID.

Then from here we're going to show a padding widget with edge insert dot only top will before.

Enter the child.

We're going to assign a text widget.

You.

Got it.

Let's dial it.

Text style font size will be 12 and font weight will be font weight bold.

And finally color will be colors dot black.

Now it's only going to show if it's allowed.

And the user has voted on that poll like this.

Orange.

Now to use the show user what I'm going to go to my hair while I'm passing it, I'm going to say show user what is true.

And then in my build voted poll section.

So we set it to true here.

And in the voted we're also going to set it to true.

And in our.

Poll section we're going to set it to.

Shallow water falls.

And in our vortex section, we've set it to true.

So I created this water section.

It's basically the same as all and posted with the same condition we used for watching.

And the only thing we're doing is showing our list using our list view builder.

If the polls have been voted on we're just basically filtering them here.

If any polls contain our current users ID adds in the voted section.

Then we just filtering those polls and saving them for ourselves and then just showing it using a list view builder.

This is the default list where we're showing it and the condition.

Now let's try running it.

Let's sign in.

So I'm going to work on this one.

And I'm going to vote for the second option.

And as you can see, data is showing confirming that I have successfully voted here.

And it's also showing in my voted section.

Now if I log in using my other user who created the poll, he'll be able to see that some progress has been made on this poll.

It's not going to show in any other tabs for him.

To wrap up, here's what we've achieved.

Votes are securely stored and updating in the database.

User can only vote once per poll, ensuring fairness.

The app now shows personalized result displaying only the polls each user has voted on.

The entire voting process is smooth and user friendly.

And that's it.

What functionality is now live and working perfectly?

Thank you for watching and I'll see you in the next video.

10. Splash Screen in Flutter

Hello everyone!

In this video we will talk about splash screen inflated.

I'll walk you through how to create a landing page with user authentication and poll creation features.

We'll start by setting up a splash screen.

Then we'll move on to implementing user login using Firebase.

After that, I'll show you how to navigate between screens based on whether the user is logged in or not.

Finally, we'll dive into creating and managing polls, including how to upload images for each user.

Let's get started.

First thing I'm going to do here is create a new folder.

I'm going to call it onboarding.

And inside it I'm going to create a new file.

Splash screen dot dot.

And I'm going to make it a stateful widget.

Let's call it a splash screen.

And I'm going to go to my auth class.

And here I'm going to create a new method that's going to return whether the user is logged in or not.

I'm going to call it user login.

And make it static.

Final user.

Get the user and Firebase auth.

Node instance stored.

Current use.

Now this is going to tell us whether the user is logged in or not.

Because if it's not now then we're going to return true from here.

Otherwise await this first.

And then we just make an else statement.

Return false.

Let's save this.

Now in my splash screen I'm going to create a new method Future of void and I'm going to call it we get to next screen.

And first thing I'm going to do is.

Future dot delete for a duration of seconds of two, because I want it to stay on the splash screen for a couple of seconds to make the UI more interactive.

And then I'm going to check is logged in and then using cloud services.

Dot user login.

Then I'm going to create a navigator dot push place a page transition.

To style child.

I'm going to pass a condition if is logged in.

Go to home screen.

Otherwise go to sign in screen.

Then I'm going to set its type page.

Transition type dot.

I'm going to replace it with.

Question.

Answer.

And this is going to be as string.

And to the route.

Going to assign false.

And if there is an error I just want to go back to my sign in screen.

So I'm just going to remove this condition and to the body of my build.

I'm going to assign it a scaffold.

And body will be centered and child will be imaged as it.

Sits and images I.

Splash screen image throat PNG.

And I'm going to set the feet to.

Work.

Box cover.

Then in my main class.

I'm going to replace this with splash screen.

Remove the redundant imports.

And restart.

We forgot to call our navigate method on init.

So let's do that.

In each state.

Super.

Let's restart this.

As you can see, it directly redirected us to our home screen.

Instead of signing because we've signed in already.

So our auth credential must still be saved in Firebase.

So I'm going to go to my sign in screen.

And I'm going to log in using another account.

And as you can see this user has already voted for this poll.

Now if I try watching again it's not going to allow me to vote.

So this user to create a poll which color do you like?

Red.

Yellow and green.

And just assign it some images.

Last image and create poll.

So our poll was created.

Now if I log in using my first user.

This poll is going to be visible to him.

And if I vote for, say, green.

I've successfully voted and it will appear in my voted section.

Now, if I try voting again, it won't allow me to wrap up.

Here's what we've accomplished in this video.

The start screen ensures a smooth transition to the appropriate screen based on user authentication.

Firebase handles user login and data management seamlessly.

Polls are created with image uploads, and user can only vote once per poll.

User can view and edit or delete their own polls in a dedicated section.

And that's it.

Thank you for watching.

I hope this helps you build your own landing page with such features.

5_state-management-with-provider

1. Introduction to State Management

Hello everyone and welcome back.

Today we'll be discussing about state management for real time data using provider.

If so, why?

State management matters.

State management is a critical aspect of flutter app development, especially when dealing with real time data.

The provider package is one of the most popular and efficient ways to manage state in flutter applications.

So what is provider?

Provider is a flutter package built on top of inherited widget system.

It simplifies state management by making it easy to share and manage application state efficiently across widget provider.

Ensures efficient state updates and scalability, making apps more responsive.

Now moving on to the key benefits of provider first.

It provides reactive updates, which is automatically rebuilt when state changes.

Scalability suitable for both small and large applications, and finally separation of concepts.

It gives the UI and business logic separate, making it easier for long term development.

Now how does a provider work?

Provider separates the business logic from the UI by creating state.

State is held in a class that notifies listener when it changes.

Second step is providing state.

State is provided to the widget tree using change, notifier, provider or other types.

Third is consuming state widgets.

Use consumer or provider to listen to and rebuild based on state changes.

Now that we know the basics of provider, let's quickly go over what we've learned about provider and provider.

Simplify state management, making it easier to share data across widget.

It uses change notifier to trigger objects and ensure widgets rebuild reactively.

It maintains a clean and scalable architecture, keeping UI and business logic separate.

That wraps up our discussion on provider.

Thank you for watching and I'll see you in the next one.

2. State Management Using Provider Part-I

Hey everyone, in this session we'll be talking about state management using the provider package.

Flutter apps need efficient state management.

Instead of using set state, which rebuilds the entire screen instead of set state, we're going to use provider.

Unlike Setstate, provider only rebuilds the required widgets in the entire widget tree.

In this video, we'll set up a provider and create a change notifier class.

And lastly, we will update the UI simply using Notify Listener.

Let's move on to the implementation.

In this video I'll be showing you how to use provider instead of Setstate.

Because when we use Setstate, the entire widget is rebuilt starting from the scaffolds or whatever your base widget is.

So instead of that, what we're going to do is use the provider class.

I'm going to show you in the dev the provider package.

Let's open this one.

You can download any version from here.

I'm going to be downloading the latest one.

So I'll go to the installing tab.

And I'm going to copy this from over here and add it to my YAML file.

Right under my dependencies.

Let's save this.

Next I'm going to create a new folder in my lib class and I will call it the provider.

Inside here I'm going to create a new class auth provider dot dot.

And I'm going to call this the auth provider class.

And it's going to extend my change notifier class which is a key part of how provider works.

So let's import this control dot from my material class.

Across.

It allows us to notify listeners whenever there is a change in the state.

Next I'm going to go into my sign in screen.

And here we have two variables show password and load.

I'm going to move this to my old provider.

And that's how we're going to handle our changes.

Because currently if we were using Setstate to update let me search for it.

And like we were doing it here using Setstate.

So I'm going to go to my provider, I'm going to make this private.

And what I'm going to do is create a better for it.

So a boolean getter for show password and I'm going to assign it my hidden show password field.

Now what I'm going to do is create a function here white set show password.

And it's going to be given a value, and inside it I'm going to update my hidden field or show password and assign it the new value.

And after that, what I wanted to do is notify listeners.

Next I'm going to do the same thing for my loader.

So I'm going to rename it to is.

Loading.

And just do the same thing for that one.

Rename it.

And I'm going to create a new variable.

And this will have the updated value and rename show password to is loading.

So now that we've created our provider let's head over to our sign in screen.

And what I'm going to do here is I'm going to wrap whichever fields need to be updated.

When Notify listener is called, I'm going to wrap it rampage inside of our consumer.

So I'm going to say control George Balzer for now.

And then I'm going to replace it with consumer.

And then I'm going to provide it.

And what provide along with high.

Let's import it provide a package.

And to the consumer we also have to specify which provider we're using.

So I'm going to say auth provider here.

Now what I'm going to do here is use my auth provider this one.

And I'm going to say show password.

And next where I'm using my show password I'm going to replace my set state with the method I created.

So also provides a choice set show password.

And I'm going to assign it the value that is opposite of the current value.

So I'm we're just going to do is come here and notify the listeners.

And replace this.

Then I'm going to go ahead and do the same thing with my isloading state here.

I'm going to wrap it inside a builder first.

And then replace it with the consumer and specify that it's of the type auth provider.

Then replace this with our provider and child.

I'm going to replace my loader with auth provider which is loaded.

Do the same here.

Auth provider dot set is loading and assign it or provided dot is loading and it's going to assign it the opposite.

Copy this and paste it here also.

Now that we've used the provider, let's say this class and call some of the extra tabs we have for now.

Inside my main class, I'm also going to have to initialize the provider.

So for that I'm going to wrap my material app.

With widget.

And I'm going to call it provider.

Now it's going to need its provider property.

I'm going to call the change notify provider and replace create with an arrow function.

I'm going to pass it the Earth provider class that we just created.

Now we've initialized the provider which we just created.

Let's save this and run our app.

Let me go over what we just changed right now.

So when we wrapped it with the consumer, a consumer will always be listening to you.

It's all provided.

And instead of calling Setstate, we used our set show password.

So what this does is when we call set show passwords and it says notify listener.

After updating the value, it's going to go to all of its listeners, which are the consumers, and it's going to see which of them are using the auth provider dot show password or other properties.

And then it's just going to update the show password property and not update the entire widget, which is extremely helpful when we usually have much larger trees with animations and pictures and rebuilding them every couple of seconds, for example, for showing a timer or a ledger or anything, it takes a lot of rebuilds and the UI may appear stuck at some point.

So instead of that, we use our auth provider for a smoother transition.

Let's check it out.

And then my password.

As you can see, my password is showing now.

But if I click on this it starts hiding.

And then if I click on it again.

Now what's happening here is that widget tree is not rebuilding the image, the text fields, the other buttons.

We have only building this text field.

And similarly if I sign in after inputting my password.

It's only going to be updating the larger button, not the entire screen.

Another way I can show you how this works.

Visualize it.

I'm going to add a debug point next to my scaffold.

So whenever I go to my.

Login screen, the widget tree is built for the first time, right?

So if I were to enter my email on my login screen or better yet, let's first go to the sign up screen and it will help you understand better.

Let's wrap this scaffold in a debug point.

Now what's going to happen here is when we click on our sign up button, we're going to be calling it twice right.

So it's two times that the widget tree will be rebuilding the entire thing from scratch or the scaffold.

So let's after adding that go to our sign up screen.

As you can see, this is the first time it's sitting because it's building for the first time.

I'm going to enter an email.

And a password.

Sign up.

So it was hit the first time when the loader started.

And then the second time when the loader stopped.

Right.

So that's how it worked on our sign up screen.

If I go to my sign in screen now and set up a debug point here.

Let's go to our sign in screen.

It's the first time.

Now, if I enter my email here and then the password, if we had been calling set state here, it would have been called twice, right?

But since we're using auth provider, let's see if the entire widget tree will be rebuilt by hitting the scaffold.

So let's sign in.

And as you can see it was not hit because only the areas that needed to be hit using the consumer were rebuilt.

So that's it.

We've successfully used provider in a flutter app.

All right.

Let's quickly recap what we've gone through.

Provider simplify state management making the code cleaner and scalable.

It provides unnecessary widget rebuilds improving app performance.

Change notifier and notify listeners efficiently.

Update the state consumer.

Allows UI to rebuild without using Setstate.

With provider we achieve a more optimized and structured flutter app.

Thank you for watching and I'll see you in the next video.

3. State Management Using Provider Part-2

Hello everyone!

In this video we will learn some more about state management.

We will replace state with provider to manage state efficiently.

Using the consumer widget will ensure only necessary parts of the UI object instead of the entire screen.

This will optimize how text fields, images and loaders work, making the app smoother and cleaner.

Let's move on to the implementation.

In the provider folder I'm going to create another file called the poll provider.

So I'll just show poll provider extends change notifier.

Next I'm going to go to my create poll screen.

And from here I'm going to copy my fields because I'll be handling the logic entirely in my provider.

So let's copy these and add them to our import the required libraries.

Next I'll also be moving all my functionalities from my UI, separating my logic from the UI entirely.

So let's add this here.

Import this.

I'm going to remove Setstate calls from here because we're going to be using provider.

Now import Fieldvalue and remove these unnecessary set states.

Also from here.

So let's go ahead and save this file.

And obviously we're going to have a few issues in that create poll screen.

So since we're working with a lot of fields and properties now I'm going to wrap the entire A scaffold with a builder and then replace it with the consumer.

Instead of wrapping sales separately and sell it, it's going to be the whole provide replace the pole provider and finally child.

Now integrating the provider in our pole screen will be very smooth.

All I have to do is a pole.

Dot and it's just going to know where to fetch the title controller from and all the other properties.

So let's go ahead and say.

And then next to my image then next to both option images I'm going to add my provider next next to my loader.

Also I'm going to use pole provider and do my submit function also.

Now it's going to know what function it needs to execute right from my poll provider class.

Remove the unnecessary imports.

Now.

Now that I've integrated my create poll screen, I also need to add it to my main.

So copy this and I'm going to replace it with call provider so it knows to listen to this provider.

Also next to code to use Notify listener instead of Setstate I'm going to call it inside my fake image where I call set state and I'm going to set it as notify listener.

And then next with the loaders where we were calling set state.

After updating the field, let's rerun our app.

With our new changes.

Next I'm going to create a new poll so I can show you how it works.

I'm going to ask a question here.

Which vegetable to you eat.

And then a couple of options.

Finally, let's say broccoli and rice like images.

Select the first image.

Now what's happening here is the entire screen for create bowl is not rebuilding, but rather only the images is updating with the updated path.

Similarly, if I create a new image here, it's only going to update this index, not the entire screen.

Finally, the last one.

Now let's click on Create Poll and we know only the loader version is being updated instead of the entire screen.

So poll was created successfully, which means our provider is working just fine.

Let's see in the question section and we can see our question here.

Next I'm going to add this functionality of provider in my edit screen.

Scream.

Now I'm going to go to my edit call screen.

And again I'm going to copy.

The functions from here.

And I'm just going to use the same poll provider.

So let's copy this and edit here as you already have an upload function I don't need this one.

Let's remove the site states.

And call notify listener here.

Same thing here.

Since we have our validate form and pick image functions here, we don't need to copy them from here because they're the same.

And we can just remove the redundant code.

And for my class I'm going to existing images here I'm going to add it here.

I'm going to remove everything else from here because we don't need it next instead of calling this on it.

I'm going to copy all of this from here and inside my call provider.

I'm going to create a new function and I'm going to call it Coid initialize call.

It is going to create a document snapshot of both.

I'm going to add all of that here.

Remove the widget.

Now to call this in my init state I'm going to first create a provider for it.

So provider instance is going to be provider dot of import it and it's going to be the type provider.

And then I'm going to pass in the context.

And I'm going to set listen to false and listen basically asking if we want to keep listening to the provider.

So provider dot initialize call and I pass it the torch pole.

Next, I'm going to wrap my scaffolds with a consumer widget again.

To replace Spencer.

Cove type pole provider.

And in here.

Pole provider and a child.

Go ahead and do the same stuff like we did before to replace it.

So dot pole provider dot title controller and replace it everywhere else.

Also the peak image function and options.

Existing images is also available in our provider.

And then the loader.

And finally the update pole.

And that's it.

We've integrated provider in an edit screen.

Let's save this.

And we're going to need to update the validate function because we're also validating images in it.

And we don't need to do that when we're working with editing it.

So let's copy this.

And I'm going to just call it from two and replace the image x and save this.

I'm going to call this function in my update call.

So call this one instead.

Save it.

Now let's check it out.

I'm going to edit this one.

And you can see it's being initialized just fine.

The images are loading and our title and options are loading.

Fine also.

So I'm going to update the text here.

And let's try and edit our poll.

And you can see the poll was edited successfully.

And we didn't need to change our images, because we've also set up that if they are already added,

we don't need to add new ones.

And that's it.

Now my entire app is using provider for managing the state.

So so to summarize, all you have to do is first create a provider class and update your data in this

class.

Then you have to wrap whatever widget you want in the consumer.

You can wrap the individual widgets like containers and text fields, or you can wrap the entire scaffold.

Finally, all you have to do is go to your main class and wrap your material app with the provider and

the provider properties.

Just tell it which providers will be updating.

So in conclusion, now our app fully uses provider, making state management more efficient with consumer

only the required parts update improving performance.

Our code is now cleaner and easier to scale for future updates.

The app feels much smoother to use, enhancing user experience.

Thank you for watching and I'll see you in the next one.

6_firebase-cloud-messaging-and-apk-generation

1. Firebase Cloud Messaging (FCM) Part-I

Hello everyone and welcome back.

In this video we will be talking about Firebase Cloud Messaging, also known as FCM.

I am going to show you how to add Firebase Cloud Messaging to your flutter amp.

We'll start by integrating FCM into the project, setting up notification handling, implementing both background and foreground and finally testing it with the sample message.

Let's get started.

So first thing we're going to do here is add FCM to our app.

So for that go to dev and search for Firebase Messaging and copy the latest version or any other version if you want.

And I'm going to add it to my public file.

Let's save this.

Next I'm going to go to my main class.

And right under my main function I'm going to create a new function and VM, set it as the entry point.

This is the same code you'll be using every time to initialize Firebase Cloud Messaging in your app.

Next I'm going to call it Firebase Messaging Background Handler.

And I'm going to be passing it a remote message.

And set it as message.

Next let's import it the Firebase Messaging.

And I'm going to make it async.

Here we have to say Firebase dot initialize app.

And in my options I'm going to set it as default Firebase options Dodge current platform, and then

I'm just going to add a log to ensure it's working correctly.

And I'm going to say handling a background message.

You'll see how it will come in handy later.

For now, let's interpolate our message dot message.

I enter my initialize app.

I'm going to say Firebase Messaging dot on background message.

I'll just call my Firebase Messaging background handler and without the parameter.

Next to handle the messaging I'm going to create a new folder in my features.

And I'm going to call it notification.

Let's call it Firebase Notifications.

And I'm going to create a new class in this Firebase Notifications service.

Dot dot.

I'm going to create the class here Firebase Notification Service.

And I'm going to set up a Firebase Messaging instance.

I'm going to call it Messages and Firebase Messaging dot.

Assign it the instance.

Next I'm going to create a new function which will return void.

And I'll say setup interact message passing the message context.

I'm going to create an instance of message and initial message.

Is going to be.

Messages.

Dot get initial message.

Now I'm going to check if my initial message is not equal to null.

Now I'm going to say Firebase Messaging dot on message open app dot.

Listen and it will be passed in event.

Next we're going to need the device notification for receiving the notification.

So I'm going to say get device token.

Each device will have a unique token so it can receive the notifications.

So I'm going to say token is equal to await messages dot get.

Chocolate.

Then I'm going to say messages dot on token refresh dot.

Listen.

to an event and assign the token.

The event which will be the new token.

Add a semicolon here.

And finally I'm going to return the token from here.

Next I'm going to create a function to request user to allow notifications.

So I'm going to say request notification permission.

Notification settings.

And I'm going to call it the settings variable.

Here I'm going to await messages dot request permission.

We're going to ask for a couple of things.

I'm going to say set alert.

We're going to need that for announcements.

Then we want to show a badge on top of our app icon in the home screen.

And then we also want CarPlay to be true.

Critical alert for any critical messages we want to send, along with provisional ones.

And finally, I want the sound to be true us.

And I'm going to check if the user has allowed it or not.

So settings dot authorization status is equal to authorization status dot authorized.

If so I'm just going to log it for now.

You can perform other actions here also.

So user granted permission.

Let's copy this.

Method.

And else here.

If provisional means some of them were allowed I'm going to say user Garnet it.

Provisional permissions finally create another as here.

And if it is denied, I'm just going to say user denied permissions.

Next I'm going to create a function to handle foreground notifications.

So I'm going to say future of void foreground methods.

I'm going to await messages dot set foreground notification presentation options.

And I'm going to say alert is going to be true.

Sound is going to be true.

We don't need much for this one.

And finally the badge is also going to be true.

Then I'm going to say Firebase messaging dot on message dot.

Listen to remote message and if message dot notification is not equal to.

Now I'm going to create a new function for this, which will be show local notifications.

And I'm going to be passing the message notification.

Now let's create this function.

For future of void the function name and it will be passed a remote message obviously of type message.

And first I'm going to create a new Android details.

And we're going to need Android notification details.

So for this you have to import a new library.

So Ctrl shift If we had dependency.

I'm going to add flutter local notifications.

Okay.

So let's import that.

And we're going to be creating a couple of things in my Android notification details which will be the channel ID.

That's what we're going to need.

And then I'm also going to need the channel name of the Android device, the channel description.

Let's set it to description.

And I'm going to need importance and importance dot high priority.

Priority will be high also.

And finally I can I'm going to need icon launcher.

So whichever icon is being used finally I'm going to need the platform details.

Is.

So I'm going to say notification details and Android.

I'm going to see the Android details.

Next I'm going to need a plugin.

So let's create an instance of that flutter local notifications plugin.

Call it that.

And I'm going to get its instance.

Let's go back to our function.

And it seems we have an error.

Oh I see a text type as remote message instead of remote notification because we're sending the notification

not the actual message.

So we need to change its type.

Let's set it to node now.

Now I'm going to go ahead and actually show my local notification.

So I'm going to say Local notifications plugin.

Add async.

Don't show.

The ID will be zero.

Title will be message title and the body will be message.

George.

Body and notification details will be platform details.

Next I'm going to create a function to initialize my plugin first.

So I'm going to say user of void and init local notification.

And I'm going to say Android initialize and Android initialization settings.

And I'm going to say app icon.

Now to initialize the settings I'm going to say Initialization settings is equal to initialize patient settings.

And I'm going to pass it my Android and to the Android.

Android initialize.

And finally I'm going to say flutter local notifications plugin George initialize and just pass it my initialization settings variable.

Let's save this class.

And then I'm going to go back to my main class.

And I'm going to change the type convert it to a stateful widget.

Then I'm going to create an instance of Firebase Notification Service.

So Firebase Notification service is equal to Firebase Notification Service.

And then I'm going to go back inside the stage of my app I'm going to create the init method.

So white init State.

Superordinate state.

And then I'm going to call my functions that I just created one by one.

So first I'm going to call the request permission because that's the first thing we need.

Then I'm going to say Firebase notification service or foreground message call that.

So start listening to any foreground messages we might receive and then set a incorrect message.

Ask the context.

Then I'm going to call my get device notification.

And after that's done I'm going to.

Get the value from it.

And I'm going to log it so I know what the value is of the device token.

And then I'm going to go to the top and let's close it from running for now.

And then I'm going to open my terminal.

And when you're working with plugins and you add a new one, something you always have to do is run, filter, clean and then update.

So all your plugins are initialized properly.

If you don't that you are going to get unexpected errors.

So just go ahead and do this ritual.

Let's run the app now.

It should work fine.

Let's go to our terminal and in my debug console.

Here we have our token for this particular device.

Now let's create a test message in the Firebase console.

I'm going to go into the build.

Or run.

And from here I'm going to select messaging.

I'm going to say create your first campaign.

I'm going to select Firebase Notification messages for sending notifications.

But if you want to include chats in your app you can also select this one.

So let's use notifications for now.

The notification title will be a new poll.

And that should be we've created a new poll.

Now I'm going to send a text message and paste the token that I copied from there.

Remove spaces.

Add it.

Now I'm going to test it on this.

Console.

And it seems we have an error.

I see what it is.

For this, you have to make sure your icon launcher is inside your android.

I'm going to go to my show Local notifications and this one.

Let's go to our Android Insider Android and App source main and then res folder inside the drawable.

We need it.

So I'm going to copy it from the first one.

Copy this.

And add it in my drawable.

So paste I need to rename it to match the same name that I've added to my drawable.

It's going.

So the reason this error is showing is because the notification is also going to have my app's logo in there.

So that's the reason it's throwing an error because it couldn't find the path to the logo.

So let's save this.

Let's rerun our app.

From my console I'm going to copy FCM token again.

And let's try it.

Send a new message.

Add it and let's test.

It was received successfully.

In our notifications we've successfully integrated Firebase Cloud Messaging into our app.

We set up notifications handling for both foreground and background states.

We create the device token for targeted messaging.

We also tested everything to ensure it works smoothly.

Now our app is fully equipped to handle push notifications.

Thank you for watching and I'll see you in the next video.

2. Firebase Cloud Messaging (FCM) Part-II

Hello everyone.

Let's continue talking about Firebase Cloud messaging in Flutter.

In this video I just want to walk through those functions again.

What they do, why we need them and how they all fit together in a bigger picture.

Each function has its own specific role.

One handles the initial Firebase setup, another takes care of receiving the messages and finally show

it to the user.

I will be going over the flow from start to finish.

Finally, when a notification comes in, Firebase Messaging triggers one of our listener.

For example, on message if the app is open or on background message if it's not inside those, we decide what to do with the data.

So let's get started.

So to start with our setup incorrect message.

This method sets up how the app reacts when the user opens the app by tapping on a notification.

The get initial message checks if the app was opened from a terminated state via a notification, and

our own messaging opened app a licence for when the app is opened from the background by tapping a notification,

so that means it's continuous.

Next we have our Get Device token.

This one fetches the unique FCM device token used by Firebase to send push notifications to this device,

and on token refresh, continuously listens to changes.

For example, a token is changed after the app is reinstalled.

Next we have our init local notifications.

This one basically just tells us to use the app icon, and that it simply initializes our plugin without

doing this.

If it's not initialized and will not set our app icon, notifications will not be displayed.

Next is the request notification permission.

This will ask the user for permissions to show notifications on iOS.

It's required to explicitly ask for permission while on Android.

Permissions are usually granted automatically.

Next is the foreground message.

By default, FCM notifications don't show automatically when the app is open.

I set foreground notification presentation options tells the FCM to still show alert search notifications

when the app is open.

One message continuously listens to notification from Firebase when a message is received, it's calling

our Show Local notification function, which we created to display the notifications when we have our

Show Local notification function.

This one defines how the notification should appear, a priority, importance, icons, etc. then finally

flutter local notifications plugin is used to display the notification on the device with the provided title and body.

So to summarize what we're doing in this class, we request notification permission, get and refresh the device's FCM token, handle messages whether the app is terminated in background or in foreground. Integrated FCM with local notifications for consistent user experience.

Let's go to our main class next.

And if we have this function for checking our background messages, this line tells the compiler not to reset this function during compilation.

It's necessary because Fvwm runs this handler in a separate isolate, which is basically a thread from the main app.

Without this, the function might get stripped out during optimization and your background notifications won't work.

This function has to be called on the top level, not inside a class.

It's called automatically when the app receives an FCM message.

While completely in the background or terminated, the remote message parameter contains the payload, such as the title, body and any other data.

And to make it actually work, you have to call it inside your run app function.

So make sure you destroy it in there.

So to conclude in this video, we went back and looked at all the notification functions we created earlier.

We broke them down piece by piece to really understand what's happening behind the scenes.

We also talked about the purpose of each function, where it fades into the notification flow from setting up Firebase to listening for messages and finally displaying it.

You should now have a clear picture of how everything connects together.

And finally we went over how notifications behave.

Depending on your app state, whether it's open, minimized or completely closed.

That's it.

Thank you for watching and I'll see you in the next one.

3. Configure Flutter App Icons and Build Android & iOS Apps

Hello everyone!

In this video we will talk about flutter app, icon set up and building APK for Android and iOS.

In this video, we'll learn how to create app icons for both iPhone and Android.

We'll use an app icon generator to make this process quick and easy.

Then we'll set up the icons for both platforms.

We'll also update the app's display name and lastly, generate a release APK for sharing on Play Store.

By the end, you'll be able to share or publish your app with a professional looking icon.

So let's get started.

So just go to your browser and search app icon generator, and all you have to do here is upload any app icon.

And it's going to provide the right size for both Android and iOS and maybe even other applications.

So this is what I'm using and I only need it for iPhone in iOS and Android.

So let's generate it.

Okay.

Then you go into your downloads and extract it.

Extract.

Now we have images for our Android and assets or assets which is for iOS and Android.

We also have multiple sizes.

You have to copy this from here and then go into your Android folder.

Then app source and main press and then you paste it inside the res folder.

And replace the ones already here because they're Flutter's default icons.

Let's open this one.

I'm going to copy this one and inside my drawable.

I'm going to paste it here and replace the default icon.

Then I'm going to go back to my downloads and app icons.

And I'm going to copy this folder.

And go back inside my iOS folder and runner I'm going to paste it here.

Replace it.

Okay.

So in my emulator you can see the icon has been updated.

So let's go into our Android.

App.

And main and Android manifest file.

And from here we're going to change our flutter course name to something more presentable.

First you have to replace it in the Android label from here.

So I'm going to college.

Fletcher poll app.

Let me run the app again and it should reflect the changes.

Cancerous.

I start debugging.

And as you can see our app name has been updated also.

Next I'm going to show you how to create a Redis build for your flutter app.

So go to your terminal and here you're going to type the command Fletcher build apk hyphen hyphen redis.

So Fletcher builds APK and then release is going to start creating your release build.

So let's go into our folder and inside our build folder outputs The journey began.

We have already begun.

Now I'm going to show you how to generate built for the Play Store.

For this particular app.

For that, you have to go to your browser and search for flutter build doc.

And under flutter documentation you're going to find build and release Android app.

First we need a keystore.

So for that.

I have to reference the Keystore for the app and you need these values.

So let's go to our project.

And inside the Android folder I'm going to create a new file and I'm going to call it key dot properties.

Now we can copy this from here and add it inside our new file.

Now I'm going to replace my store password with anything.

I'm going to call it 123.

And same for My key password.

Now I need a file location.

Let's save this file first and go back to our documentation.

Here we have our command for getting our location on Mac OS.

You can run this one, but since I'm on windows, I'm going to copy this one and run it.

Let's have the same password here and do it again.

I'm going to say Paul and course and course here again part of U.P. and Punjab PK set it to no here.

Here I'm going to enter my own name.

Iwp in Punjab.

And finally add.

Yes here I learned the same password here and again.

So now Keystore file has been created in this path.

I'm going to go to this path and copy the file from here.

And I'm going to paste it inside my Android folder.

Let's open that in the file explorer.

And I'm going to paste it here.

Let's go back to our flutter project.

And now we have our file here.

Let's copy it and replace the file.

Let's save this.

And I'm going to go back to my terminal.

And here I'm going to run Pubget first to get my dependencies.

Then I'm going to run flutter build app builder and release.

This process will take some time as it's generating a release version of the app.

The app file is what we will upload to the Play Store.

It's the format required for publishing Android apps.

Once the build is complete, every listed A or B is generated in the output folder.

So let's go into that folder.

Inside the build folder app outputs, we have our new bundle folder.

It has a folder.

This is the file we're going to be uploading to the Play Store.

So we created the file, generated the Keystore, added its path and build the app release AaBb.

So to conclude what we've learned today, we successfully created an integrated app icon for both Android and iOS.

We ensure the icons are correctly placed and replaced, keeping our app looking professional.

We also updated the app's name and permissions for proper functionality.

Finally, we built a release ready APK and adventure, making the app ready for distribution.

That's it!

Thank you for watching and see you in the next one.

7_offline-data-with-sqflite-and-firebase-sync

1. SQFLite Setup and Data Storage

Hello everyone.

In the last module we worked on Firebase Cloud Messaging, in which we learned how to set up push notifications

using Firebase.

And we also covered APK generation.

This included building and signing the APK for both Android and iOS.

Next, let's talk about something super useful for flutter mobile apps SQLite.

It's a simple serverless database engine that doesn't need any setup or configuration.

What makes it best for flutter is that it runs inside your app, which means no internet and no external

server required so app can work offline and store data locally.

Let's get started.

So the first thing I'll be doing today is adding the libraries I'll be using to my pubspec.yaml file.

Under here.

First I'm going to add Connectivity Plus, which is going to tell me whether the internet is connected

or not.

so I can store data offline.

I'm going to use this version 6.0.3.

Next we have shared preferences.

And I'm going to use it's version 2.2.3.

And then there is SQLite the actual database package.

I'm going to be using 2.3.2.

And the last one is a path I'm going to be using 1.8.3.

Let's go ahead and save this file.

And then I'm going to go into my features folder.

I'm going to create a new folder called the helper folder is where I'm going to be creating my database

class and probably the Sharedpreferences helper also.

So let's create a new file called Database helper dot.

Dot the class called database helper.

Then I'm going to create a singleton class, which basically means that only one instance of the class will be used across the entire apps lifecycle.

So that means there are no memory leaks.

So I'm saying instance equal database helper.

Dot internal.

I'm going to be creating the internal.

It basically means that internally that instance will be used within the class.

So factory database helper constructor.

And then its private instance is written.

So for the internal I'm going to be saying database helper George internal.

Okay.

So here I'm going to create static database.

And underscore database.

Variable that is used to access the database.

It will have the direct connection to it.

So let's import the database class.

So next I'm going to create the create db method.

And then I'm going to assign the resulting database to this private database variable.

And then I'm going to create the future database.

And then I'm going to create the init db method.

And it's going to return a future of database.

Underscore init db.

And I'm now going to pass it.

Anything else I'm going to say.

Final DB path A path to the local databases.

And get database path.

And then I'm going.

To create a path to my particular DB.

So I'm going to say join.

DB paths with my database path so it can be anything.

I'm going to say my database dot db dot is essential.

So it knows that creating a database.

So this join is the path we imported.

Okay.

Next I'm going to create a return statement of Await open database.

I'm going to return it the paths also the version which is one for now.

If we make any changes, we end up changing the version so it updates the changes.

And then we have the Oncreate method.

So the Oncreate method is the part where we create the actual tables.

So I'm going to say await db this db dot execute some SQL query.

So here I'm going to pass my SQL query.

You have to put three in the starting and three in the ending.

I'm going to say here you have to follow the syntax exactly.

Create table.

Any table name I'm going to say false table.

And then I'm going to say an ID right.

And for string in this database we have text.

So ID text primary key.

And then we have title with also text database data type.

And then we have creator ID.

And it's also going to be of text data type.

And then created at.

That's going to be an integer.

Then we have total words which is also an integer.

And then for the last one we have a status like so.

And that's it.

That's how you create a table in the database.

So we have just created our pulse table.

Let's go ahead and also create our options table.

For each pole we have multiple options right.

Like three options.

So that's why I'm going to do.

A weight db dot execute.

I'm gonna say.

Create table all options.

And then press it.

Then we have first variable is id let's say integer which is gonna be a primary key that auto increments.

Next we have our.

College.

Which is gonna be a text.

I'm gonna copy text so I can paste it easily.

I'm gonna say name is also text.

I'm gonna say.

Image URL is also text.

And then we have words which is an integer.

And for the last one we have a foreign key.

That's gonna be call ID.

Which references.

All stable.

And its ID column.

And if we delete the stable column we say delete cascade.

So all the related data is deleted.

And there we have it.

This is how we create two tables in the database in SQLite.

So next now that we have created these let's assign this database to this database variable.

So I'm going to say.

Future.

Of database.

Getter.

For that database I'm going to make an async function of it.

And I'm saying that if this database instance is not null Then return that.

Otherwise, we're gonna go ahead and assign something to it first by the database.

So I'm going to say await in stages.

So this is returning database right.

So they're both the same.

Um, I'm gonna make this note now.

And from here, we also return this database.

And that's it.

That's how we create the database.

Next I'm going to go ahead and create a function to insert data into the database.

So I'm going to say future of void in Z.

Ball with files.

The option files right?

So I'm gonna say the following are required a string which is call ID.

This is the data we'll be inserting.

So we need to have it required.

String.

Title requires string.

ID.

Then we have required date.

Time variable.

Named as fg date.

Then we have the required list of string, a list of string for option names, and then also for the

auction files, the images I'm gonna say required list of file, which may be null.

And I'm saying option files.

Let's import this.

So next I'm going to go ahead and say get me the database.

Now that we have the database we're going to go ahead and insert our database data into the database.

So I'm going to say db dot insert which is totally easy.

You just provide the table name and then its value which will be passing as a map.

So we'll be telling it this is the column name.

And we want this data to be inserted against it is a polite right.

And then we have title.

So I'm going to say title.

Then we have our creator IG.

So let's say creator ID.

Then we have created at.

So I'm going to turn the created ad because we saved it here as an integer, not a datetime variable,

because that doesn't exist in the SQLite world.

So I'm going to convert it into an integer string that will be reverting back when we fetch data from it.

So I'm going to say millisecond since epoch.

And then I'm going to provide it with total words which by default are going to be zero because we're just creating it now.

They there aren't gonna be any votes for it.

And then we have the status.

So I'm going to say status here is add.

So the status functionality here is that it tells us when we save data into the database that we're adding the database or updating the database.

So when we fetch the database to send it to Firebase we'll know what function we need to perform with it.

And then now that we've added this feature we also need to add data to the option table.

So I'm gonna say for since these are lists I'm going to be using the for loop.

I is less than option names dot length.

I plus plus.

So I'm going to first get the file path.

I'm gonna say option Machine bias of I th index which can be null dot.

Get me its path.

Otherwise let's just return an empty string.

So I'm gonna go ahead.

I'm going to say db dot insert into false underscore.

Call underscore options table.

A map with the following values.

So it's going to be the call ID.

So I'm gonna say call ID.

And then I'm going to have its name city name.

Sorry option name of its index.

And then I'm going to pass it the Image URL.

So I'm gonna save this file path that we just patched.

And then we have our words which is still gonna be zero by default.

So yeah, that's how you create this data tables.

And then insert the data into the tables.

I'm gonna move into the poll provider class and where we created our submit poll method for submitting data to Firebase.

I'm also going to create a method that submits data to our local database instead.

So I'm going to say future void check.

So.

Possible.

When I have to pass the following parameters to it such as string creator id and build context the same as submit code.

And then I'm also going to make it async.

Now I'm going to try and validate the form validate form context.

And if it's not validated return from here.

Otherwise we can move on.

Final option names.

Equal option.

Name.

Controller.

Dot map.

E dot text.

Dot Tolist.

Now we have our option name.

Let's get our option files or the paths to our files.

So I'm going to say option files equals Equal option images.

Now I'm going to try and insert this into our database.

I'm going to say database helper dot insert Paul.

So Paul ID is going to be some random ID for now since we haven't created a document on Firebase.

So I have this function right here that creates a new ID every time.

So I'm going to use that here.

I'm going to say generate random ID for title.

I'm going to get title controller dot text creator ID field is already filled in.

Then we have created ad would be date time.

Dot now.

And this is about all the things we have.

Now I'm gonna show a message saying the data was submitted into our local database.

So I'm gonna say scaffold's.

Messenger dot context.

Dot show Snackbar.

Const Snackbar.

Content is going to be text.

No.

Internet.

Poll saved locally.

I'm going to wrap this in a try catch block.

So if there is an issue we can handle that.

So I'm going gonna say.

Control dot there with try catch.

So I'm gonna copy this.

And show a message here also.

And then I say fails to save or locally control F.

So that's about all we need to do here.

I'm gonna go ahead and save these classes.

Next I'm going to go find references of this submit code.

It was used in this class.

So I'm gonna replace this with check submission.

Possible to see if our method is working fine and the data is being saved locally.

Right.

So I'm going to go ahead and connect my device.

Now I'm going to run my code.

F5.

I'm going to try and save the poll.

Let's see fruit.

Let's check the pictures first.

So I have the mango.

A strawberry.

And then basically I'm going to go ahead and create this poll.

So it says no internet.

And the poll was saved locally successfully.

So in this video we learned how to successfully implement local database with SQL Lite and store data into a local database.

2. Checking Internet Connection Availability

Hello everyone and welcome back.

In our last video, we learned how to store data in a local database using SQL fly.

In today's video, I'll be telling you about internet connectivity in flutter.

In this video, we'll be learning how to setup connectivity handler for internet monitoring before saving data.

The app will check for an active internet connection.

If online data will be stored directly to Firebase.

Otherwise it's going to be stored locally.

Let's get started.

So the first thing I'm going to do is head over to my features folder and create a new folder called services.

And then I'm going to create a new file in it called Connectivity Service.

Dot dart.

Then I'm going to go ahead and create my connectivity.

Service class.

Then I'm going to create a final connectivity variable that is going to receive from connectivity.

Constructor.

So for this connectivity you're going to have to import the Connectivity Plus package.

I've already imported it in the version 6.0.3.

You can import any version.

And I'm going to then import it within the class.

Package connectivity plus dot dot.

And I'm going to create a controller for it.

When I say controller equal stream controller.

So we can check throughout if the internet turns off during the apps life cycle.

So it's going to return a type of bool.

Then I'm going to create a constructor for my connectivity service class that is going to.

Return if the internet service starts.

So dot listen delegate.

That will keep listening to the internet service.

So I'm going to create a check actual internet method myself.

Control dot create method.

It's going to be a method of type future void.

And I'm also going to make it async.

Then I'm gonna add a try catch block first.

It's important here because if it does not detect the internet being on, it might throw an exception.

So I'm just going to say controller dot add false in that case so that it tells that the internet is

off.

Otherwise what we're gonna do is gonna get the results, say await internet address.

We're going to look it up on the internet, any, any website we can look up.

And if it returns and if it returns some results, we can say the internet is on.

And if it doesn't, means the internet is off.

So I'm gonna say controller dot add.

Results.

Dot is not empty.

And results.

Zeroth index dot row address dot is not empty.

So if both these are true, then the result will be true.

And then I'm also going to create another function.

This function will be started whenever the connectivity service constructor is called.

And when I want to say return a bool when I create a check now method.

And I'm also going to implement the try catch here.

It's going to be mostly the same logic.

When I say return false from here.

Shift alt f.

And then I'm going to say async Thing first.

So the error is gone.

Final result is equal to await internet address stored lookup.

Again, the same logic that we try to look it up.

And if it works it means the internet is on.

This logic right here is going to return the boolean value.

So I'm going to directly return it to here.

So let's put the return statement here first.

And there we go.

Now we've also created a controller that we need to dispose of.

So there are no memory leaks.

So I'm going to say void dispose.

Underscore.

Controller Earlier George Gough.

And there we have it our connectivity service class.

Now I'm going to go into my poll provider class where I created this text submission possible method.

So basically now we have to make sure that if the internet is on we actually save the data to Firebase.

Otherwise we're going to save it to our local storage.

So I'm going to check if is online then await connectivity service dot channel.

And now the if condition if is online.

We're going to use this method here because we're online.

So we can just submit the poll to Firebase.

I'm going to say await submit poll.

I'm going to pass it the creator ID along with the context.

That we passed as parameters here.

And if this condition is not true then you're going to wrap this block in an else.

Then I'm also going to show a message.

Because if the data is saved online we have to say.

Saved successfully.

Let's go ahead and run our program and see if it works.

So let's go ahead and create a new ball while the internet is on.

Let's say the same one as we did last time.

I'm going to select the images first.

Window.

Thank you.

So I'm going to go ahead and save it.

So the poll was created successfully.

We don't need to add this here because it's already showing a message in the submit poll function.

So now we have this I'm going to delete it.

And then I'm going to close my internet and then create it offline.

So it says no internet.

And the poll was saved locally.

So in this video we learned how to successfully implement a connectivity handler for internet monitoring.

And we also check internet connection before storing data.

This approach ensures that no data is lost during offline usage.

Thank you for your time and I'll see you in the next one.

3. Syncing Offline Polls to Firebase When Online part 1

Hello everyone and welcome back.

In the last video, I showed you how to store data based on your internet availability.

In this video, I'm going to show you how to auto sync data to Firebase from your local database.

We'll be learning how to auto sync Unsync posts to Firebase when the internet comes back on.

And I'll also show you how to use shared preferences to track a posting status.

Let's get started.

So the first thing I'm going to be doing here is creating my app for shared preferences.

I'm going to create a new file and say prefs underscore helper dot dot.

So I'm going to create the class sync prefs.

And I'm going to create a function that is static.

And I'm going to say it's future void test.

And it's gonna set and sync status based on the boolean value that we receive.

I'm going to make it async since it's future.

So I'm going to create a variable that can hold the reference to the shared preferences.

Dot get instance.

I'm going to import it.

Import package.

Shared preferences.

Now that we have its instance I'm going to set its value.

I'm going to say props.

Dot set Boolean A type of Boolean value.

You can name this key anything.

When I say has unsynced poles.

And then I'm gonna just put a comma here.

And this is how you set the unseen status.

And I'm gonna create another function that tells us.

If we have any data to be synced to Firebase.

So I'm going to say as and sync false.

So it's gonna be first getting the prefs.

Shared preferences dot get instance I'm going to make it async.

And then I'm gonna return prefs.

Dot get bool.

Here we use set bool.

And here we're gonna use get going.

So I'm going to pass the key I use here.

And if it doesn't exist it's just gonna return false.

So now that we created our Shared Preferences class, I'm going to go back into my poll provider class and into my check submission possible method.

Now where we're adding our data to the firebase I'm gonna be setting await prefs dot set and saying status to false.

I'm going to import this and then just copy it.

Since we've already submitted data to Firebase we're going to set it to false.

And here since we're adding it to our local database I'm going to set it to true.

So here it's going to be true.

And then I'm going to be creating a new function called Future Void.

Try sync local false.

It's basically going to be checking frequently if the data is there in local database to be synced to Firebase.

And if the internet is on it's going to go ahead and save it there.

So I'm gonna say has unseen.

Equal await sync prefs dot has.

And simple.

This is going to tell if there is any data to be synced.

And then I'm also going to get files is online.

If it's online then you can go ahead and save it to database.

So I'm going to say connectivity service.

Dot Tot check now.

So first the condition has to be if it has unsynced holes and if it is online.

If both these conditions are true, then means if both these conditions are true, then it means we're good to go.

I'm going to get a database instance from database helper.

And then I'm also going to get the local polls from the database instance get all polls.

Now I'm going to be creating this method in our database helper class.

So this method is basically going to be returning all the data that is currently stored in a database.

Is.

And it will return it in a form of a list.

So I'm gonna say get all false.

Media async.

So first is gonna get final DB equals await database.

And then it's gonna return await db dot query on the false table.

Now we're gonna go ahead back into our pricing poll function.

And we're going to start using a for loop.

Because there can be a lot of polls saved in database while we were offline.

So I'm gonna say for poll in local polls if there is any data in our local polls.

Let's await this.

First you're going to create a folder in database in the firebase.

So you can store the data in the document.

I'm going to say Firebase Firestore dot instance dot collection.

And our collection name was false in Firebase.

And then I'm going to say dot.

So it's going to create a new doc for.

Then I'm going to get the doc's ID.

Hold of.id.

And then I'm going to say final title equal all title.

So I'm going to copy this.

One for creator ID.

Creator id and the other board status.

And then we also have to do it for.

Total votes.

So I'm going to say here total votes.

Okay.

So next I'm going to go ahead and put a condition here that if the status or status is equal equal to add then you can just go ahead.

And create a created at variable.

I'm going to get date time dot from millisecond since epoch.

And I'm going to say

That since we turned it into an integer value when we were storing it into our local database, we need to now convert it back into a DateTime variable.

And now I'm going to say Firebase Firestore dot instance dot collection.

And the collection name is false dot doc.

Since we created the doc I'm going to pass it the doc ID I'm going to say Paul ID dot set this map.

Now I'm going to pass it all the required fields that are Firebase.

Paul ID title.

Then there is ID created ID.

Then we have creative edge.

Total votes.

Total votes.

And then we have options.

So for options if we didn't convert it yet I'm passing an empty list for now.

But we're going to do it later.

So since we've added it now, we can move on.

Now I'm going to create a constructor for our poll provider class.

And before that I'm going to create a function called Start Connectivity Sync.

And I'm going to say final service equal connectivity service.

I'm going to use this service.

Dot internet status.

The internet status.

Stream I'm going to go ahead into connectivity service and add it real quick.

It's going to tell us if the internet is available or not.

So I'm going to say stream of bool.

Get internet status.

Stream.

From controller dot.

Stream.

Save.

And now that's available I'm going to say dot listen.

Let's make this a thing.

And I'm gonna say if.

This event is online for understandability.

I renamed it and I'm gonna say if it is online, then await.

So I think localhost the function that we just created.

And I'm going to call this on the poll provider constructor.

So whenever poll provider is called it's going to start listening to the internet activity.

So I'm gonna say start connectivity.

Let's go ahead and save our file and run the project.

Let's go ahead and sign in.

I'm gonna move to Austin Positive and I'm going to turn my internet off.

And then I'm going to try to create a new goal.

So I'm going to name it group as we did in the previous videos.

And then I'm going to add the pictures.

And then I'm gonna add their names for example mango strawberry.

And for the last one it's a Kiwi.

And I'm gonna create the poll.

So it was saved offline.

Let's go ahead and turn our internet back on.

As you can see the poll group was created.

But it's options are not showing because we only passed an empty list.

For now we're going to add it later.

So that's all for today.

In this video we learned how to automatically detect and sink and sink polls to Firebase when we come back online.

And we also learned how to track poll sink status reliably using shared preferences.

Thank you for watching and see you in the next one.

4. Syncing Offline Polls to Firebase When Online part 2

Hello everyone and welcome back.

In the last video I showed you how to auto sync new posts to Firebase when an internet connection is detected.

In this video, we'll be seeing how to sync poll options with polls to Firebase and use shared preferences to track poll scene status.

Let's get started.

So last time we left our poll options as an empty list before sending it to Firebase, this time I'm going to try and add data to it from our local database.

So first thing we're going to need is go into our database helper class and create a new function in it that returns us a list of all options.

So I'm going to say future of list of a map.

String dynamic.

And I'm going to name it.

And I'm going to name it Get Options for all.

It's also going to be receiving a poll ID on the basis of which it's going to fetch its data from the local database.

So I'm going to say final DB get db instance and then return db dot query.

First we're going to have to tell it the table name.

So our table name was poll options.

And then we're going to pass it the where argument that it needs to check where poll ID is equal to this.

And in where arguments we tell it to replace the question mark With this argument.

So the syntax needs to be exact so it can work.

Let's go ahead and save our class and go back into our pod provider.

Now I'm going to try and create a variable say options equal await db dot get option for course.

Instead of this fall ID which is a new document in our database, I'm going to get the ID from this.

I'm going to provide ID next within our if condition.

I'm going to create a new list called Final list of Map string dynamic.

And I'm going to name it Firebase Options.

I'm going to leave it empty for now.

And then I'm going to create a for loop for looping through all our options in our options list.

So I is less than options dot length and i++.

Now first I'm going to get the option from our options list.

So I'm going to say option of I.

Then I'm going to get the name from our option variable.

So I'm going to say option name.

Then I'm going to get the image path.

Because in our local database we stored the image path and not the actual image.

So I'm going to say image URL.

And then I'm going to get the image file for my image path.

So image file is equal to.

Image patch.

And then I'm going to get an image URL like we did in our submit code function.

I'm going to say.

Image.

So image file.

Then policing the document one from Firebase and then the AI index.

Then I'm going to add this data into my Firebase option list.

So I'm going to say add a new map.

First thing is going to be name.

And then we're going to have store the image URL.

So image URL.

Save option.

Of works.

And if it doesn't exist, I'm going to pass zero by default.

And then all I have to do is instead of passing this empty option list, I'm going to pass it.

Firebase options send a list of Firebase option instead of that empty list.

So let's go ahead and run the project.

So I'm going to go ahead and sign in.

Go to my posterior Street section, and I'm going to go offline first and then create a new poll.

The same one we usually do fruit.

Then I'm going to select the pictures.

Strawberry and kiwi.

Let's name them.

And then let's save the poll.

So now internet and the poll were saved locally.

So let's turn the internet back on.

So as you can see the poll was saved successfully to Firebase.

So in this video we learned how to automatically detect and sync unseen polls with the poll options to Firebase when the internet comes back online.

And we also learned how to track polling status reliably using shared preferences.

Thank you for watching and see you in the next one.

5. Updating SQFLite Records & Syncing Updates

Hello everyone and welcome back.

In the last video I showed you how to auto sync.

New polish file based on an internet connection is detected.

In this video I'll be showing you how to auto sync updated data to Firebase from SQL flight.

We'll be learning how to auto sync updated calls and call options to Firebase when online, and use shared preferences to track all sync stages.

Let's get started.

So these are my status of Add.

I'm going to start checking the conditions for if status is equal to update.

In that case do the following.

So in this part also I'm going to first create a list of map that will give me the options.

So I'm going to also call it the Firebase options.

And then I'm going to create the for loop.

I is less than options for length and I.

And then I'm going to say final option is equal to options of I going to fetch that option.

Then I'm going to extract its name.

So option of name.

Then I have the image path.

So option of image URL.

Next I'm going to create a string.

And I'm going to store the URL in that image URL.

So in the scenario we simply extract the URL.

So when we're updating a poll, we're going to need to first check if the image path.

Dot starts with.

Some string.

So I'm going to check if it's an HTTP string.

Because when we click on update the image are going to be auto filled already.

So if we align this there's a chance that the image will not be out of phase.

Or we could be updating the image.

So if the image path has HTTP that means we're using the Firebase image.

Otherwise we're using a new image that was uploaded by the user.

So I'm going to check if image path is starting with HTTP.

And if it is I'm going to say image URL is equal to image path.

Otherwise we're going to be extracting the URL using our upload image function.

So I'm going to say final image file is equal to File image path.

And then I'm going to say image is equal to await upload image.

Image file.

And that's how we will be extracting the URL.

Then I'm going to go ahead and add these options into my Firebase Options list.

So add a new map step is going to be the name.

So name.

Then we're going to have the image URL.

So image URL.

And then the last thing is going to be the words.

So I'm going to put here option or words.

And if it's empty by default zero.

Now I'm going to go ahead and add this into the firebase.

Then I'm going to go ahead and add it to my firebase.

Firebase Firestore dot instance dot collection of polls dot up documents.

Document is gonna be poll ID.

You're going to see that when we add it to.

Our local database.

We're saving the document ID instead of generating a new ID like we did not add function.

So I'm going to pass a map here.

I'm going to be updating the title.

This is going to be title and then options.

It is going to be Firebase Options.

Now that we've added our sync logic, let's go ahead into our update function.

Our update function.

And I'm going to check here if it is online before my trial.

I'm going to say final is online is equal to await connectivity service dot check.

Now and then after all the options are filled in I'm going to check before adding to Firebase.

If it is online or not.

So I'm going to wrap this in a try catch block up to this message.

I'm going to say if is online.

Then perform this.

Otherwise Wise, we're going to be performing the LS as F, then I'm going to add my LS logic here.

I'm going to say final db instead at database server await db dot update call with options.

So this function doesn't exist yet.

In database we're going to go ahead and add it there.

Hold on I'm going to say future void update all with options.

Is going to be provided the following parameters.

It's going to need colliding.

Then it's going to need a new title.

Then it's going to need a list of map of string All options or we can call it update it.

Okay.

Then we can just go ahead and add an instance.

DB equals await database.

Then we're going to check if there are existing calls in database first.

So I'm going to say await PB query is going to check the table calls where.

ID is equal to question mark.

So the question mark will be replaced with where arguments.

I'm going to provide it call it.

And then I'm going to check if.

Existing calls Though it is empty.

Await DB dot z.

False.

And then here is the map.

It is going to be call ID.

Right.

This is going to be new title.

And then status is going to be update of course because we are in the update function.

And then in the else condition we're going to say await db dot update table.

False.

With the following map.

Title.

Title You is going to be new title and status, which is going to be.

Provided by where?

ID is equal to this and where arguments are colons.

So basically what we did here is we checked if a poll was already edited and added to the database,

and if it wasn't we're going to just add a new one.

If it was we're going to pass the else condition.

It's going to update in our local database.

And then we're going to go ahead and within the else condition we're going to delete any old poll option

that were already in the database.

Is this all going to say poll option.

And then I'm going to say where.

Poll All ID is equal to something in where eyes is polite.

And then for your option in updated options that we passed as parameters, I'm going to say await db dot in z.

Table is call options.

And the values are.

ID.

And then name.

You toggle.

Option name.

Image URL option Image URL.

Last reward.

So I'm going to say zero and that's it.

That's our update function.

We updated the whole options in it.

Also going to go back and pass it all the required options.

So call ID.

And title controller dot text for the new title.

And then updated options.

I'm going to wrap this with the if condition.

I'm going to say if if online then go ahead and store the URL.

Otherwise we're going to get the image URL.

Otherwise you're going to get the image URL like so I thought I was going to be storing it.

So now I'm going to go ahead and run it.

I'm going to go into my polling tab.

Turn my internet off and edit it.

I'm just going to add one in the end.

Put in the first option and I'm going to change the picture to Kiwi.

And the polls are saved locally.

Then turn it back on.

And here you can see the poll was updated successfully.

So today we learned how to apply changes to our data in the SQL database.

And we also learned how to enable real time syncing of updated data to Firebase.

Thank you for watching and see you in the next one.

6. Deleting SQFLite Records & Syncing Deletion Logs

Hello everyone and welcome back.

In the last video, we learned how to add using updated data to Firebase and also update it in SQL flight.

In this video, I am going to show you how to perform deletion of data that is stored in a SQLite database and how to automatically delete posts from Firebase when the user comes back online.

Let's get started.

So the first thing I'm going to do here is going to my show post class and into my delete goal function.

And before saving the data to Firebase I'm going to check if we are online or not.

So I'm going to say await connectivity service dot check now.

And then I'm going to wrap my saving to Firebase Logic with an Then if.

Condition.

Controls all round with F is online.

Otherwise.

All shift F else.

Is going to create a db instance.

So I'm going to say final db is equal to database server.

I'm going to create my function in my database helper class that will be called.

In Z call for deletion.

Basically we'll be creating it so we can share the data to our local database when the internet is off, and then when it comes back on we're going to sync that, delete the data to Firebase.

So I'm going to say DB is equal to database.

Await it.

And then I'm going to say await db dot in Z.

I'm going to pass it the table's name.

Then I'm going to pass it a map of values.

The only main thing we need to pass here is the id all id the rest of the things I'm going to leave as empty.

So I'm going to say title, but I don't have to assign anything to title.

And then I'm going to say ID and empty.

And then I'm going to say created.

And I'm going to assign it datetime.now.

It will work fine if you don't assign it anything either.

So I'm going to say since epoch, and then I'm going to assign it a total votes zero, since I can assign it an empty string.

And then I'm going to say status, which is the important one.

I'm going to assign it delete, and I'm going to close it.

Now that we have our insert for for deletion function, I'm going to save the class and close it.

I'm going to call that function here await db dot z call for deletion.

Pass it the policy.

And then I'm going to just show a message here using my snackbar.

I'm going to say.

No internet.

All will be deleted Native when you come back online.

So that's it.

That's all you have to do in this class.

I'm going to go ahead and save it.

Then I'm going to go into my poll provider class.

Also, I'm going to need to add another thing.

Set my sync preferences to true.

Because we do have data we need to sync when we come back online.

So I'm going to say true.

So now I can go back into my poll provider class here in my dry sinkhole function I'm going to add a new condition.

Else if status is equal equal to delete I am going to.

First check in my Firebase document snapshot.

Hold job is equal to await Firebase Firestore instance dot collection.

In the polls collection dot check.

For this poll ID or the doc number is going to be the same as poll ID.

So let's say poll I g dot get.

And then I'm going to see if this poll IG or doc exists.

Then we're going to try and access the images that were saved against our collection.

So I'm going to say list of dynamic options is equal to poll dog of options.

Otherwise an empty list.

And then I'm going to run a for loop for it for option in options.

So I'm going to check if options image URL is not null.

I'm going to try and delete it from Firebase first before deleting the actual document.

So I'm going to say string.

Image URL is equal to option of image url.

And then string file path.

Get the firebase from Firebase.

So uri dot decode.

Cool.

I'm using this to get to the path and say uri dot pass.

I'm going to pass it my image URL He is going to try and get there using path segments in my image URL and then checking last where element dot contains holes because that's where our images are stored in our database.

So I'm going to say if it's not possible then or else do nothing.

So now that we have our path to our image, let's go ahead and try and delete it from our file.

So I'm going to say file path is not empty.

Await Firebase storage dot Firebase Firestore but Firebase storage.

Dot instance dot reference to this Firebase path file path dot delete.

It's easy to delete once you have the full path.

So now I'm going to go ahead and try and delete my document from Firebase dot instance dot collection.

Form.

Calls.

Dot doc all of ID dot delete.

And then once we've deleted it from our firebase we're going to go ahead.

And when we're outside our if condition once we're done syncing everything from our local storage, we're also going to try and delete it from our local storage.

So I'm going to save for VAR call in local polls.

Await DB dot delete poll.

I'm going to create this function in a second.

First we have to pass it the id.

So I'm going to go to my database and create this function.

It is very easy to delete data from a local database.

I'm going to say delete poll passage string.

Call it.

Async.

Then I'm going to get first the db instance equal await database.

And then I'm going to say await db dot delete poll options.

Where.

Quality is equal to this and in the way I am going to pass it the whole item.

So we have to delete the options first because deleting the reference might cause an error.

So delete.

When we're trying to delete.

Hello everyone and welcome back.

In the last video we learned how to add node options.

It's going to work for the references.

If it doesn't find the reference.

Firebase Cloud Messaging in which we learned how to set up push notifications using Firebase.

And we also covered APK generation.

This included building and signing the APK for both Android and iOS.

Next, let's talk about something super useful for flutter mobile app sets.

It's a simple serverless database engine that doesn't need any setup or configuration.

Done syncing all our data.

What makes it runs inside your app, which means no internet and no external server.

So your app can work offline and store data locally.

Let's get started.

So I'm going to run my app and close my folders tab.

Let's log in.

Go into our postage section.

I'm going to try and delete my profile.

So it says it was deleted locally.

Let's turn the internet back on.

And as you can see it was successfully deleted from Firebase.

So in this video we learned how to implement functionality to remove data from our SQLite database.

Then we also learned how to auto delete posts from Firebase when the internet is turned back on.

So that's all for today.

Thank you for watching and see you in the next one.

8_google-maps-integration-and-proximity-filtering

1. Google Maps Setup

Hello everyone and welcome back.

In the last module, we explored the SQLite database and learned how to store data offline.

We also covered Autosync, which included saving data offline and syncing it to Firebase once we're back online.

Now let's move on to integrating Google Maps.

We'll learn how to set it up using the Google Cloud Console, and also how to configure the necessary dependencies and permissions in our flutter project.

Let's get started.

So first thing I'm going to be doing is going into my Google Cloud Console.

Make sure you're in the current project that you're actually working on.

So I'm going working on Flutter Course.

And I'm going to go into my API services credentials.

Here I'll be generating a new key.

So create credential a new API key.

I'm going to copy it.

Then I'm going to go into my flutter project and the Android folder app, it's source main folder and into my androidmanifest file.

Then.

I'm going to be adding the following tag under my application tag like metadata.

Android name is equal to com dot Google dot Android Geo dot API key.

Then I'll be mentioning the value Android Value is going to be the API key.

We just call I'm going to put a slash and then close it here.

Going to move this metadata tag.

Below these three.

Next we're going to need to add the following location above our application tag.

So I'm going to say uses.

Permission.

Android.

Its name is this.

So I'm going to say Android dot permission dot internet.

Next.

The next permission is going to be uses permission Android Name.

Android dot permission.

Dot access.

Fine.

Location.

I'm going to copy it.

And the last permission here.

Is gonna be access first location.

One more will be.

Access.

Background.

Location.

So these are the permissions you're also gonna need.

Also add this slash going to save this file and close it.

Next we're going to go into our iOS folder runner.

and the info.plist file.

We're going to add a new key here.

A key.

And then I'm going to add the file name and location.

When in use make sure it's CamelCase usage description.

Close the key and then provided the string value.

So the string is going to be.

We need your location.

To find or near you.

And then we're going to close a string.

Close our info.plist.

And then we're going to head into our app Delegate Swift here I'm going to import Google Maps.

Google maps.

And then I'm also gonna underbool add the following.

Service.

To provide API key.

Then here I'm going to go ahead and copy my key again.

So key copy it and add it here.

After setting up our Google Maps in our project, I'm going to go back into my console and I'm going to enable the following services.

First is going to be Google Maps SDK for Android.

This one.

Make sure to enable it from here.

It's already enabled for me.

Then I'm going to go back and enable maps SDK for iOS.

Enable that.

Then I'm going to enable geocoding.

API.

Enables.

And the last one is.

Places API.

Enable this also.

And after we're done enabling this I'm going to go back into my region credentials.

And then I'm going to edit the key I created.

Make sure to set the application's restrictions to null and API restriction to restrict key and enable all the APIs that we just created.

So I'm going to say maps SDK for both Android and iOS.

And then I'm going to enable geocoding API.

And then the last one is places API.

After enabling these all you can go ahead and save it.

And that's how you set up the Google Cloud Console and the flutter project.

So in this video we learned how to set up Google Maps through the Google Cloud Console, including API setup and key restrictions.

We also learned how to add the necessary dependencies and permissions in our flutter project.

Thank you for watching and see you in the next one.

2. Displaying Google Maps

Hello everyone and welcome back.

In the last video, we configured Google Maps in the Google Cloud Console and set it up in our flutter project.

In this video, I'll show you how to display a Google map in flutter, and also how to enable other features in it, such as zoom gestures.

Let's get started.

So the first thing I'm going to be doing here is going into my pubspec.yaml file and adding my dependencies.

The first is Google Maps Flutter using its version 2.5.3.

Then I'm going to go ahead and add my geolocator.

11 .1.0 and the last one is permission handler.

We'll be using the permission handler to request permission for location.

So 3.1.

Save the file.

Then I'm going to go into my home screens folder and add a new file manuscript.

Dot dot.

New class.

I'm going to call it map screen.

First thing I'm going to do here is create my Google map controller.

And I'm going to call it the map controller.

Next create a variable for storing our latitude and longitude.

So I'm going to say Selected position, and by default I'm going to pass it a latitude and longitude of the Google headquarters, which is 37 point.

4219 and longitude is -1 to 2.

And from here I'm going to return a scaffold.

A bar question ever.

And then I'm going to pass it the title.

Pick Location.

And then for the body, I'm going to pass it the Google Map widget.

I'm going to say Google Map.

Initial camera position is going to be camera position.

Target dot select.

Target the selected position.

And I'm going to add a default zoom of 40 so it's not too far away.

After that I'm going to provide it.

On map created which passes a controller.

I'm going to give it my map controller.

Update it.

Then we're going to be passing the markers which show our current location on the map or whichever location

we select.

So I'm going to say markers in marker.

I'm going to pass it random fields of marker which is going to be named selected.

So the current one.

And then I'm going to pass it the position.

Which is going to be our selected position.

Remove this.

On tap I'm going to add a function.

It's going to be providing us the lat long, and I'm going to take that and update our camera accordingly.

So I'm going to say set state I selected position has changed to the lat long.

Then I'm going to add a floating action button.

After my Google map in my scaffold a floating action button.

And the column.

I'm going to provide it.

Main exercise.

And its children.

Floating action button is going to be.

Navigator.

Told pop.

Context.

And I'm also going to pass our selected position.

I'm going to customize it at the background color of colors dot orange.

And then also a child which is going to be an icon.

Set.

So I'm going to call this map screen in my create whole screen.

And then add another elevated button here grab this with arrow widget.

Then I'm going to copy this elevated button because of the same design a size box.

And then I'm going to add the second elevated button.

And it's going to be.

Choose.

Location.

And on its own quest.

I'm going to remove this.

Final.

Lat long.

Result is equal to.

Away.

Navigate to dot.

Push.

And the route is going to be to a material page route context.

And then I'm going to pass it my map screen.

Let's make this assume.

Let's go ahead and run it.

So it seems we have a greater mismatch issue.

Now this is not a big thing.

All you have to do is go into your Android app build operator.

Going to change it from this to 20.

A Kotlin version.

Going to upgrade it to 1.9.0.

Save it and let's try it again.

Let's sign in.

Try and create a new poll.

Choose location.

As you can see, the map is being rendered correctly and we can move the pointer around on click.

In this video we learned how to display Google Maps, and we also added features such as location marker

to improve interactivity.

So that's all for today.

Thank you for watching and see you in the next one.

3. Implementing 'Locate Me' & Sending Location to Firebase

Welcome back everyone.

In the last lesson, we learned how to display Google Maps and add markers to it.

In this video, we'll take it a step further by implementing a lookup feature to jump to the user's current location.

We'll also cover how to send location data to Firebase.

Let's get started.

So the first thing I'm going to do is create the lookup feature.

So I'm going to create a function here future Y mutate me.

And I'm going to make it async.

I'm going to get the status.

I'm going to say permission dot location.

Let's import this first.

George request.

So it tells us if the location permission is granted or not.

So I'm going to say status dot is granted, in which case go ahead and locate it.

So position.

Position is equal to.

We're going to use the Geolocator package.

So get current position.

And we want the accuracy to be very high.

So I'm going to say location accuracy dot.

Five.

Then I'm going to get the lat long from this position.

So the new position is the lat long.

Position.

Dot latitude.

And position dot longitude.

Then I'm going to go ahead and assign the new position to my selected position variable.

So it updates the map marker.

I'm going to say new position.

Then I'm going to access the map controller.

And I'm saying animate the camera to the current position.

So I'm going to say camera update dot new lat long zoom.

So it's animated in a way that doesn't look bad.

And new position and zoom is still 15.

And if it is not granted we're just going to go ahead and show our message.

Scaffold messenger dot of context dot show Snackbar.

Snackbar Snackbar and its content is going to be a text saying location.

Permission denied.

So now that we've created our Look at Me feature, I'm going to create a new floating button.

That on click of it it goes to the user's current location.

Location.

And icon is going to be my location.

Let's save it.

So now that we've got a current location we're going to try and save it to Firebase.

So I'm going to go into my poll provider class and I'm going to create a new variable in here that long.

And when saving it to Firebase I'm going to add a new variable here or a new page.

And I'm just going to pass it the last name.

And when we are verifying our form I'm also going to add a new condition here that checks if location is null.

The user needs to select the location first.

So I'm going to say equal equal to null.

So a message here location is required.

Then to assign the position to the lat long variable I'm going to go into my create false screen class.

And where I'm passing it back here in the results, I'm going to access it and save it to the provider.

Results not equal to null.

The poll provider.lat long variable is equal to geo point.

And for the latitude I'm going to say result dot latitude.

And result.com.

And that's how we assign it to a lat long variable.

Now let's go ahead and run it.

Also in my map screen I forgot to add the hero tags here because each floating button is to have a unique hero tag to uniquely identify it.

So I'm going to say locate me.

And then copy this and add it to the second floating button also.

So I'm going to say confirm location.

Save this also.

So I'm going to go into my create poll section.

And choose location.

So the map is being rendered.

Let's try locate my feature.

So I jumped right to my current location.

Let's finish this up.

So in this video we learned how to enable real time location access on maps improving user experience.

And we also stored users location to display polls relevant to their current area.

So that's all for this video.

Thank you for watching and see you in the next one.

4. Displaying Proximity-Based Polls

Hello everyone and welcome back.

In the last video, I showed you guys how to submit user location to Firebase and add an interactive location feature to Google Maps.

In this video, we'll be seeing how to utilize real time location data to dynamically display nearby polls, and how to leverage your locator to accurately calculate the distance between a user and the submitted false location.

Let's get started.

So the first thing I'm going to do here is go to my show polls class and try and get the user's current location.

So I'm going to create a position variable.

I'm going to name it user position.

Import it.

Then I'm going to create a function That will fetch the user location.

So fetch user location.

I'm going to put it in a try catch block, since it might throw an exception if it's unable to access the user's location.

So I'm going to say catch anything here.

I'm going to print it for now.

And we're getting location.

And here I'm going to access the location by the position variable.

So await Geolocator dot get current position.

After getting it I'm going to update our user position variable going to assign it.

Then I'm going to go ahead and call it in the init state.

Fetch user location.

Then where I'm who is showing all my polls.

I'm first going to check if it's close to the user.

I'm going to use a five kilometres distance for proximity.

But we're going to check here if the user.

Position variable is equal equal to null.

Then just.

Show.

Then just show a circular progressbar.

Then after we're done fetching our data, I'm going to create a double next distance in meters.

I am going to assign it 5000km.

So now I'm going to call it.

All false.

And then I'm going to create a new variable called false.

And I'm going to filter the assign false.

Going to filter the all false placed where.

Data of this poll.

Created as a map.

And then filter it to data position from a firebase.

Is equal equal to null or.

Data of position is not a geo point.

Then you can go ahead and return false.

Otherwise you can create a geo point.

And pole position equal to data of position.

Then I'm going to get the distance and check if it matches or not.

stage distance.

Use the Geolocator package to calculate the distance.

I'm going to provide it the distance I'm going to say user position dot latitude.

And then user position longitude.

Here I'm going to provide it the pole that we're currently using.

So I'm going to say pole position dot latitude.

And then pole position dot longitude.

Then in the end I'm going to return distance less than or equal to Next distance in meters.

If it's true, it will return true.

Otherwise it will return false.

I'm going to make it to list.

So that's it?

That's all you have to do to calculate the distance.

So another thing I'm going to be changing here is going into my create class.

And anywhere I created a row of elevator buttons I'm going to say main axis alignment axis alignment

or sensor.

Height into it.

Save this.

And now let's run our program.

So I'm going to log out of here and log in with another user.

So I can check whether the polls are shown based on proximity or not.

Sign in.

I am going to create two new goals.

One is going to be furniture.

Chair.

Table.

And then we have like this.

Okay I'm going to choose any random images for this.

And I'm going to choose is very far away from our current region.

So I'm saying here and then save it and create.

I'm going to create another poll.

I'm going to call it season.

Say.

Winter.

Autumn.

Then spring.

And look at you.

I'm going to choose.

Is very close to my user.

So it says.

There and then create form.

So now I'm going to sign out and sign in with my original user.

As you can see the season hall is being shown while the furniture one is not because it was not in proximity

to this user.

So in this video, we learned how to utilize real time location data of the user to display polls that

are close to the user's current location.

And we also integrate the Geolocator to calculate the distance accurately.

That's all for this video.

Thank you for watching and see you in the next one.

9_version-control-with-git-and-github

1. GitHub Desktop Installation and Setup

Hello everyone and welcome back.

In the last module, I showed you how to display Google Maps, access the user's location, and save it to Firebase along with some extra features.

In this module we'll be exploring GitHub.

Starting with GitHub desktop.

And I'll also be showing you how to download and configure GitHub desktop.

Let's get started.

So I'm going to go to Google and I'm going to search for GitHub desktop.

Download GitHub desktop.

The first link.

I'm going to download it for windows.

Let's wait for the installation to finish.

So it's finished downloading.

I'm going to run the exe file.

Let's see.

The installation is finished.

Now I'm just going to go ahead and sign in.

It's going to redirect me to its website.

Going to input my credentials.

Then my password.

Sign in.

So continue.

And it's going to ask me to open GitHub desktop.

And that's it.

That's how you set up your GitHub desktop.

So in this video we learned how to setup and configure GitHub desktop and completed the initial setup and sign in with GitHub desktop account.

So that's all for this video.

Thank you for watching and see you in the next one.

2. Creating Repository and Pushing Changes to GitHub

Hello everyone and welcome back.

In the last module, I taught you guys how to download and configure GitHub desktop for your windows laptop.

In this video, we'll be learning how to create a new repository in our GitHub desktop and publish it on GitHub, and also how to make changes and push those changes to GitHub.

Let's get started.

So to create a new repository for our existing project, I'm going to go into file and add local repository.

And I'm going to choose the path to my project.

So I'm going to say.

Select folder Add repository.

And then it's going to warn me that this directory doesn't appear to have a repository.

So I'm going to say create a repository instead.

And then I'm just going to go ahead and initialize this with the readme and create repository.

So all my changes are committed to main already.

So I'm just going to go ahead and publish it.

Publish.

So as you can see, our directory was published successfully.

Let's open our flutter app.

And in the repository folder.

So as you can see our git repository is initialized.

Let's go to GitHub and check if it's there.

So GitHub.

As you can see it's available here with all of our files already committed.

So now I'm going to make a change in our project.

And I'm going to show the changes reflected here in our GitHub desktop.

So here I'm just going to change the string I'm going to say create a new poll.

Save it.

Let's go to GitHub desktop.

As you can see this is how a new difference Different shows.

This was the old one, and the green one is the new one.

So to push it to our main GitHub, I'm going to add a commit message here.

Updated app bar.

And then all you have to do is commit one file to main.

But before committing make sure you fetch it.

So if there are any changes in our remote repository, we can view them.

This is especially useful if you're working on large projects, because there can be multiple developers working on the same branch.

If there are any changes, they will be shown here.

There were no changes in our remote repository.

So now I'm going to just commit this file and commit it.

You also have to make sure to push it to a remote repository.

So that's it.

That's all you have to do to send your changes to your remote repository using GitHub desktop.

So in this video we learned how to use GitHub desktop to create and publish our repositories.

And we also learned how to push new changes in our project to our remote GitHub repository.

That's all for this video.

Thank you for watching and see you in the next one.

3. Cloning Repository and Creating Branches

In the last module, I showed you guys how to create a new repository in GitHub desktop, and how to commit and push changes to a remote GitHub repository.

In this video, I'm going to be showing you how to clone a repository in GitHub desktop from GitHub, and also how to create new branches in GitHub using GitHub desktop.

Let's get started.

So to clone a repository I'm going to go to my GitHub desktop.

And it's going to show me all my repositories that are currently linked with my GitHub account.

So this is the app I want to clone.

I'll click on this button and I'll be cloning it using github.com.

And I'm going to choose a path here in my D drive.

In my projects I'm going to create a new folder letter app and I'm going to create it in there.

Let's clone it.

And that's it.

That's how you clone a project in flutter using GitHub desktop.

That is super easy.

There is no complications in it.

Now I'm going to try and create a new branch in my app.

So I'm going to say new branch.

Let's say back end.

Create branch.

And then you have to publish it to your remote repository.

So as you can see cloning a repository is super easy.

And creating a new branch is also very easy.

And to view any history of commits we can go here.

And as you can see there was the initial commit.

That sent all my files to the remote repository.

And then there was the updated bar that I changed the string of this app bar.

So in this video, we learned how to use GitHub desktop to clone other repositories that are linked

to your GitHub account, and also how to create new branches and publish it on the remote branch.

That's all for this video.

Thank you for watching and see you in the next one.

4. Merging Branches and Discarding Unwanted Changes

Hello everyone and welcome back.

In the last video, we explored GitHub repositories and branches, and we also learned how to view our

commit history using GitHub desktop.

In this video, I'll be showing you how to merge branches in GitHub desktop and also how to revert unwanted

changes in GitHub desktop.

Let's get started.

So here I'm going to make a change in my app.

And then I'm going to push the changes to the GitHub.

And then after that I'm going to be merging the branch.

So the first thing I'm going to do is go to my map screen and let's just edit this string.

So location permission is denied.

Save the file.

And then I'm going to go to my GitHub desktop.

And it shows me that this one string has been changed.

I'm going to commit it update it location.

Feature region.

And now we're just going to commit my file to back end.

And then I'm going to push a region.

So in the history we can see our updated location string.

And in my main backend branch.

So I'm in my main branch right now.

I'm going to choose a branch to merge into main to get the updated code.

I'm going to select backend so it will tell you that there is one commit from backend that will be merged into our main.

So create a merge commit.

And then all you have to do is push origin.

That will send it to our remote repository.

And that's it.

That's how you merge a branch using GitHub desktop.

So next I'm going to show you how to revert a change.

For example I worked on a project and there are some things I do not want to send to GitHub.

Right.

So I want to revert those changes.

So say I'm here, I say pick a location, save it.

And then when I go into my GitHub desktop this is the change right?

And if I want to say I don't want to make this change anymore I right click and discard change.

Are you sure you want to discard this change?

And I say yes.

And in our VS code this change is also reverted.

So in this video we learned how to merge branches using GitHub desktop and also how to discard our local

changes in VS code environment from GitHub desktop.

So that's all for this video.

Thank you for watching and see you in the next one.

10_subscription-management-with-stripe-and-paypal

1. Creating UI for featuring Poll

Welcome back everyone.

It's great to see your dedication to learning.

Making it this far shows your commitment.

In the last module, we explored the ins and outs of GitHub, including how to setup GitHub desktop, create repositories, and push code to remote branches.

In this module, we shift our focus to payment integration.

In this session specifically, we will learn how to allow users to promote their posterior posts for increased visibility and also how to monetize that engagement through micropayments using PayPal.

Let's get started.

In this video, I'm going to show you how to add a featured button to our posted posts.

So I'm going to go inside my page whole card and add a new variable as a parameter.

I'm going to say bool.

Show feature button equals to false inside that.

And after setting it I'm going to go to my show.

Postage goes.

Here.

And when we're calling in our billfold card.

I'm also going to pass.

Show feature buttons.

I am going to say true.

Then I'm going to go into my Bill's hole card.

And here.

Where I'm showing my edit and delete buttons I'm going to show the feature button.

I'm going to copy this.

First I'm going to check is show feature button true.

Then you can go ahead and show this.

Otherwise just an empty container.

Now I'm going to set it stone to.

First I'm going to delete that and add going to be adding my PayPal logic here.

Later I'm going to turn this to feature and the icon is going to be.

Some.

Purple.

Five inches outline.

And this is both of these are orange.

You can have that.

So I'm going to turn it to deep orange.

Same for this one since the star has to be in our app colors.

So that's it.

That's all you have to do to show our features button.

Save it and run our file.

Okay.

Let's login.

And in our posted section we can see our feature button.

I'm going to turn down the size of the text a little.

Let's set it to 12.

Copy this and also set it for our edits text and our text.

And it should work fine.

Save it.

That's it.

That's all you have to do to add a featured button.

So in this video, we designed our posted post screen by adding a new feature button.

Later we're going to add our PayPal logic against that feature button.

So that's all for this video.

Thank you for watching and see you in the next one.

2. PayPal Setup

In the last session, we created a feature button for our posted polls.

In this session, we'll be setting up PayPal integration for that feature and also handling deep links to support dynamic navigation within the app.

Let's get started.

So for integrating PayPal into our app, I'm going to go into Android App Source main and then the Android

manifest file.

I'm going to add a new metadata tag under my activity here.

I'm going to say copy this paste it.

And for Intentfilter Android name action.

And this is going to be replaced with view.

Then the second category is going to be Android intent category dot default.

And then add another category here.

That's going to be browsable.

And then below the category I'm going to add a data tag.

Android scheme.

Is equal to a link of any choice.

You can name this anything, but make sure it's the same one throughout the app.

So after that I'm going to say Android host on PayPal success.

Basically, what we're doing here is setting up our deep links for navigating to the PayPal without

going outside app to the browser, staying within our app instead.

So that's why we're setting up these deep links.

And then this one was for our success case of PayPal payment, and this one is going to be our failure

case or our cancel case.

So everything is going to be remaining the same.

Just our cancel is going to be changed.

So we can just go ahead and save this file.

Now then I'm going to close it.

Go inside my iOS folder runner and Info.plist.

Here I'm going to add a new key open array like this one.

I'm going to add it below this one I'm going to say key.

F bundle.

U r l types and key disclosure.

Make sure all the spellings are correct.

Close the array first.

You have to be careful here because if there are any changes, you won't know that they are what that issue is directly.

You're going to get a bunch of run around errors.

So close that another key in here.

Make the key forward going to say C bundle.

Up.

With an S URL schemes.

Close this key.

Create a string with the name the same as you select it in your Android directory.

String.

And then just go ahead and close the array.

That's it.

That's all you need to add to our Info.plist class.

I'm going to save it and close.

Next I'm going to create another class inside a new folder in the features.

Let's create a new folder.

Payment.

And then create a new class in it payment services dot dart.

And then create the class.

Payment services.

Then inside this the first thing we're going to need is our app links for navigation.

So app links is equal to app links.

So of course we need to import it.

Import package.

App links.

Dot.

Dart control dot.

The semicolon is missing.

Now I'm going to create a new method.

Static void handle.

Deep link.

I'm going to make it async.

And then I'm going to create a URI.

Initial URI.

I'm going to name it.

And then I'm going to say await app dot.

Get initial link.

If initial link of URI is not equal to null.

Process link.

And I'm going to pass it my initial URL.

And now I'm going to create this method myself.

Is going to accept this initial URI.

If URI dot host initial URI.

Is equal.

Equal to PayPal success.

Remember how we added it?

PayPal success in the Android file?

For now I'm just going to print payment successfully.

Success.

As if it is payment.

Cancel.

Canceled.

Then I'm going to go ahead and finish my handle.

Deeplink.

I'm going to say.

Airplanes.

Dot uri link stream.

Dot.

Listen.

And this is gonna be a URI.

And then process link uri.

This is going to create an own error here for now.

Don't actually use it.

That's it.

That's how you create your deep links for navigation within our app without actually needing to interact with the browser.

In this video for integrating PayPal into our application, we first created deep links for navigation.

So when we pass the link for PayPal, it isn't opened in the browser, but rather it's open within our app.

So that's all for this video.

Thank you for watching and see you in the next one.

3. Implementing Subscription Using PayPal

Hello everyone and welcome back.

In the last videos, we learned how to set up deep links and added a feature button to allow users to feature their port.

In this video, we'll explore how to display a PayPal payment sheet and allow users to make a payment to feature their ports.

Let's get started.

So to integrate PayPal in here, I'm going to create a payment sheet first in my payments folder.

I'm going to say new file.

PayPal dialogue dot da.

So for this you're going to need a method instead of a class.

So I'm going to say future string initiate PayPal payment.

And we're going to pass it our build contact.

And then an amount of how much we're going to charge the user.

And then the last thing is our shipping amount which is optional.

Import it.

Then I'm going to use the payment service.

Dot handle Deeplink that we created for navigation.

And then I'm going to create a complete a completer is basically a method or let's say a controller that lets you return some data in this case string in the future.

So our payment through PayPal will take a time in the future before returning any result.

So I'm going to use the computer here.

Now I'm going to say Navigator George Bush.

And this I'm going to pass it material page.

Route.

And the builder is going to have some context.

Let's say use page path.

Going to import it here.

Package.

Select splitter.

And I'm going to import it to dependencies.

So now that I've got the payment sheet, I'm going to set sandbox mode to true.

Then I'm going to set the client ID here.

You need to pass your own client ID.

And then we also have to pass the secret key.

Then we also have to add the return URL which we set up in our androidmanifest file.

Dynamic.

Polls.

App PayPal.

Success.

And then I'm going to.

Going to set here the cancel URL, which is cancel.

After that you have to initiate the transaction.

First thing you have to pass it is a map.

In the map I'm going to tell it the amount that is required, and then in the amount they also have to pass an entire map, which is going to contain a list of details such as total amount that we pass in our parameters.

And then I'm going to set the currency.

To USD.

Then we have the details.

And in the detail you basically have to mention Subtotal, which is just the amount.

And then the shipping is gonna be shipping amount because you have to pass something here.

You can't just leave these fields empty.

They're mandatory.

And shipping discount is gonna be set to zero.

Then after we done telling about our amount, we need to set the description.

I'm going to say premium subscription.

You can set anything here any random string.

And after we're done with our transaction we have to pass our node.

Contact.

As for any questions.

Then on success is mandatory.

Also it's basically going to give you a map of parameters.

Async.

For this async, we need the completer that we created here because it's going to tell us later right.

So I'm going to say payment service store I'm.

Going to set it to empty I'm going to go to my payment services class and create a string PayPal error message and set it to empty.

Save it.

Also make this static.

Now we can go ahead and access it.

And I'm going to get the message from my string parameters going to get the payment ID this is probably the most important one in appearance because.

Image ID is what you usually store in your database when you keeping a record of your payments.

And then I'm going to say completer dot complete.

Basically it's going to return it to us, the payment ID.

Then we have to handle our own error case.

There are a bunch of errors that can be possible in making a transaction.

So the most basic one is your transaction fails.

So first I'm going to get the error message in a new variable.

Error dot toString.

I'm going to check if error message contains these particular words, because this is the message you're going to receive from the error transaction refused in these letters.

Exactly.

Then I'm going to set my payment services dot Apple message to because our card was basically declined here.

That's what the transaction refused means.

So I'm just going to say your payment was declined.

Please try a different card.

And in my else if I'm going to check if the error message contains internal.

Server error because it's entirely possible that the server is down.

So I'm going to copy this.

Set the message to.

PayPal is experiencing issues.

Please try again later.

And if both of these are not the cases and we still have some sort of error, I'm just gonna set it to some generic message that we were unable to process your payment.

Because there are a lot of other scenarios that are possible here.

Then I'm going to set the completed dot complete and I'm going to send it.

The message failed.

And the last case here is on cancel.

We also receive some parameters here.

So I'm just going to go ahead and set it to complete and failed.

And after you run this you can just return completed dot future.

So that's it.

That's all you have to do to set up your initiate PayPal payments.

Going to remove the path here.

Okay.

So I've set up my client ID and secret key.

And also make sure to set sandbox mode to too fast when you're moving your app to production.

This is just for testing purposes, so let's save it.

And here I'm just going to go ahead and call the payment for PayPal.

Initiate PayPal payment.

Just have to pass it the context.

Shipping amount is going to be $5.

I'm going to set the shipping amount to zero.

So like 0.00 that's it.

That's all you have to do.

Let's try and run our program now.

Let's head over to our posture section.

And try and feature our poll.

As you can see, it is navigating within our app because of the deep links we set up the login.

So as you can see we're being charged a $5 amount.

Let's continue.

And this should finish up the payments.

So the payment was completed successfully.

We haven't shown any message for now.

So that's why there is nothing showing.

So in this video we learned how to integrate PayPal payment into our application and how to make a one time payment for in-app purchases.

That's all for this video.

Thank you for watching and see you in the next one.

4. Sorting and Highlighting Featured Polls

Hello everyone and welcome back.

In the last video, we learned how to make payments using PayPal to feature a poll.

In this video, we'll explore how to update poll status in Firebase after payment is confirmed, and also how to prioritize feature polls by placing them at the top of the field.

Let's get started.

So the first thing I'm going to do is set my feature status in Firebase.

For that I'm going to go into the poll provider class.

And here I'm going to create a new method.

I'm going to make it static.

So I can access it directly.

I'm going to call it feature poll.

And I'm going to be passing a poll ID to this so we can update.

It is featured variable.

I'm first going to fetch the data from Firebase Is to give a snapshot and.

Instance.

Collection of calls.

And where call id.

Is equal to.

This for line.

And then I'm just going to fetch the first one.

And.

Then I'm going to check if my query snapshot is not empty.

Then I'm going to get the document reference from it.

Commit the query snapshot, dot docs, dot paste and its reference.

Then I'm just going to say doc wrap dot update.

I'm going to set it as a map.

So I'm going to set is featured to true by default because I'm going to be calling it only when the payment is success.

And then I'm going to add another variable expiration date.

And I'm going to set date time dot now.

And I want it to expire after ten days.

So I'm going to set that as the duration.

Duration to and I'm going to set its days to ten.

And that's it.

That's all I have to do to feature my poll in Firebase.

I'm going to save it and go back into my.

PayPal payments.

Now I'm going to check if my result is not equal to fail in the case of cancel or error.

Right.

So if it is not fails I'm going to access my poll provider.

And then I'm going to say feature poll and just provide it the poll ID.

I'm going to access my poll ID in the beginning around here I'm going to say call ID is equal to data call ID.

Go back.

Now I'm also going to go ahead and show a message displaying our success.

Show, snack bar and snack bar.

I'm going to say congratulations.

This poll is now featured.

And in the case of error, if result is equal equal to failed.

I'm also going to display a message.

Let's copy this.

And instead of a normal message in a string.

I'm going to show the message that I saved in my payment services.

George Pepall error message.

And and I'm just going to return it from here.

Now, before displaying this.

I'm also going to check if the goal is featured or not.

Each feature is equal to data of is feature.

Now I'm going to check is featured here.

If it is featured then you can just go ahead and show a star icon to know that it's been promoted.

I can.

I can start.

Start.

Outline.

Otherwise, just show an empty container.

And I'm going to set its color to color dot orange.

And then where I'm checking to show my feature.

I'm also going to check here if not of each feature because if it's already featured we don't need to show a feature button for that.

So now that we're all set here in our build poll card, I'm going to go into my build All polls.

And here, after filtering it on the basis of location, I'm also going to check first.

If it is featured then it should be shown on top right.

So I'm going to say false start short.

I'm going to be passing it two lists A and B.

And then first I'm going to get data of list A a dot data.

As a map of string and dynamic.

Then I'm going to get copy this.

And for my B list I'm also going to get each data.

A data of is featured.

If it is equal equal to true then store one against it.

Otherwise store zero.

And I'm going to be doing the same thing for my B list.

Get if it is featured, then one, otherwise zero.

And then in the end return b dot b feature dot compare to a feature.

And basically this is just going to sort the list before showing it.

Semicolon here arch is f.

Save it.

Okay.

So I'm going to go ahead and run this.

Destroy feature this fall.

Continue.

We should finish up the payment process.

And as you can see, the pole was featured and we can see a star icon.

Now I'm going to log out.

Let's see if it shows his feature to another person.

As you can see, that pole is featured with a star and it's appearing at the top of the page to other users.

So in this video, we learned how to show our featured posts at the top to promote them better and how to check Mark them in our firebase to ensure that payment is accurately tracked and verified.

That's all for this video.

Thank you for watching and see you in the next one.

5. Auto-Unfeature Polls After a Time Limit

Hello everyone and welcome back.

In the last video, we learned how to feature a post to show them on top of the field.

In this video, we will implement validation to check all expiration dates prior to displaying them, and also automate status updates to mark posts and run featured after a predefined duration.

Let's get started.

So to configure a poll I'm going to create a function in my poll provider class.

It's actually really simple.

I'm going to create a static.

Feature void and feature poll.

You just have to pass it the poll ID.

Async.

And then all you have to do is get a Firebase Firestore instance.

Church.

Collection of polls.

And then you get the doc ID, because our poll ID is the same as the doc ID, so I'm just going to access that document through it.

And then I'm just going to go ahead and update.

It through a map.

First I'm going to set is featured.

To false.

And then I'm going to set expiration date to null.

Because I'm going to be checking that if expiration date is null it means the poll is not featured yet.

Save this.

Go back into my show polls class.

Now here in the beginning of my Build Gold card.

I'm going to be checking the expiration date.

I'm basically going to be accessing the timestamp.

Expiration timestamp.

And then get the data from the Firebase expiration date.

Then I'm going to convert this timestamp into a date time variable.

Expiration date.

George to date.

Now that we have our expiration date I'm going to check if this expiration.

Date is not equal to null.

And also if this date time.

George.

Now, dirt is before this expiration date.

And we have to apply a note here because if it's true, we want it to be false.

And if it's false, we want it to be true.

So it can actually accept both the conditions.

Then I'm going to set the each feature to false.

And I'm also going to call my Unfeatured poll function from my poll provider class.

Just pass it the poll ID and that's it.

That's all you have to do now.

All the polls will be shown if they are not expired.

Let's save this and run it.

Let me show you from the firebase.

This.

This poll is featured right.

And its state expiration date is 15 and today is 16.

So it should unfeatured this poll and it should show a feature button in the posted section.

Let's see.

Let's go into our posted section.

And as you can see it's providing me the option to feature this poll.

And let's see if it's updated our Firebase.

So expiration date is null and this feature is false.

So in this video we learned how to automatically and feature polls after a certain time to keep the content fresh.

And also how to check expiration dates to make sure only the featured polls are shown on top of the page.

That's all for this video.

Thank you for watching and see you in the next one.

6. Setting Up Cloud Functions for Setting Up Cloud Functions for Stripe Payments

Hello everyone and welcome back.

In the last videos we learned how to feature a pole using PayPal payments.

Now we'll be learning how to use stripe to create subscription plans.

In this video, I'll show you how to set up Cloud functions and also how to create local functions for testing.

Let's get started.

So the first thing you need to do is go to Firebase Documentation.

And here you see the command to initialize Firebase in your app.

We'll be initializing it so we can add Cloud functions.

So all you have to do is run these Firebase init command.

So you can go ahead and press Y here.

And then we have multiple options to select from.

But since we want to create cloud functions we'll select functions using the spacebar.

And then hit enter.

Then it will inquire about the project type, I'll be selecting an existing project since we already have one.

Now you can go ahead and search the project that you will be using here.

So this is the Project Director course.

I'll be choosing and then hit enter.

So now it's started setting up our functions.

So here it's asking about the language we will be writing our cloud functions in.

I'm going to be working with JavaScript.

So that's what I'll select here.

Select no for yes links.

And yes for our npm because we definitely need that.

Okay so here our project is set up for cloud functions.

Let's go into Visual Studio Code.

And here we can see our Cloud Functions folder has been created successfully.

So we have a couple of files here.

The important one is index dot JSON.

And we also have package.json.

The index JSON file provides us with Firebase a couple of predefined functions.

And on request is used for creating the HTTP cloud function like this function.

Let me uncomment it.

This is a predefined function called HelloWorld.

The first thing you need to do here is run the project.

So for that you have to go into your functions folder.

So CD functions.

And then you run the project.

So you say Firebase serve.

So this command will run our local project that will use our live database.

Okay.

So here you can see our cloud functions are finished executing.

So one of our predefined endpoint is this hello world function that's used for testing.

And here we can see it's URL.

So I'll copy this.

And you may also notice that this is running on the localhost 5000.

So I'm going to go to my postman and create a Get request there.

And this will send us a response.

So endpoint has sent a response.

Let's try hitting it again.

So we're getting hello from Firebase which is our response from this HTTP request of us.

So in this video we learned how to set up Cloud Functions in our project.

And also how to develop and run local server side functions for testing.

That's all for this video.

Thank you for watching and see you in the next one

7. Publishing Stripe Cloud Functions part 1

Hello everyone and welcome back.

In the last video, I showed you guys how to set up Cloud Functions in a flutter project.

In this video, I'm going to be showing you guys how to create backend functions that securely retrieve the list of available subscription options and also publish that function on localhost.

Let's get started.

So here I have already created a couple of functions to make payments through stripe.

And I've also created a project and assigned two prices against it.

One of it is for monthly payments which is 200.

And the second one is for yearly payments which is 500.

So for accessing the prices we're going to go to use the stripe API.

I'm going to go to.

Stripe API I reference.

And here.

Stripe provides a document.

Let me search it.

Prizes.

Select this.

Now you have to find the price for product API.

So under Products and Prices you can see that this is the code that strives to provide us to get the prices against a particular product.

So I'm using this code here in my function and stripe prices dot list.

And then you just provide the product with its ID.

This will retrieve the prices for both plans.

And then you can just go ahead and return those prices if the status is 200.

And they've also wrapped it inside a try catch block for extra security so it can catch exceptions, if any.

So now I'm going to run the project again using Firebase Serve.

Because here as I showed you I've created a new function besides Hello World.

And this function helps receive the subscription plan.

I receive the URL.

Let's copy it.

Now let's check if this URL is functioning correctly by creating a request.

And here you can see that I've successfully received both payment plans in the form of an array.

In this video, we learned how to fetch subscription plans from the backend and also how to execute functions in a local development environment to simulate real world backend behavior.

That's all for this video.

Thank you for watching and see you in the next one

8. Publishing Stripe Cloud Functions part 2

Hello everyone and welcome back.

In the last video, I told you guys how to create cloud functions.

In this video, we'll learn how to implement backend type functions to create customer update and retrieve subscriptions and add card details, and also how to publish the function on localhost for testing.

So here I've created four endpoints for the subscription plans.

This is an already created function of HelloWorld.

So the first one is to fetch the subscription plans.

And in our scenario we have the monthly plan.

And the other is the yearly plan.

Let me show you in the stripe dashboard.

So in the project section.

Open that project and we have them here.

As you can see one is the monthly plan and the first one is the yearly plan.

So this pain point is basically for fetching the prices using the project ID, and we get the price in response.

And the second endpoint is add customer card.

We just provide it the email and the name and the card token in its payload.

The name and email are used to create the customer.

It first checks if there is already a customer in stripe with these credentials.

If so, it doesn't create a new customer.

Otherwise it just goes ahead and create a new one.

After customer creation, we have to associate a card with the customer.

So for that we receive a card token in the payload.

And in that token we have the card details.

It contains the card number, expiry, CVC etc. and here the card gets attached to the customer.

So to review this endpoint create a new customer if it doesn't exist and attaches a card to it either way.

And the third endpoint is the subscribe one used for making the subscription.

And in this we require the customer ID.

Along with the price ID.

And then we first check if the customer is already subscribed.

If so, we just update the older subscription by removing the older item and replacing it with a new price plan.

And in the second scenario, if subscription, say does not exist against that customer, we just create a new subscription for them.

In the last endpoint in these cloud functions is for fetching the subscriptions that we have created against any user.

And here we're using the stripe API subscription dot retrieve to which we pass the subscription ID.

So these are the four endpoints that we've created.

One is to get prices and customer card, subscribe and retrieve subscription.

Let's test it first.

Firebase serve.

It's finished running.

So as you can see it's provided as a separate URL for each endpoint.

Basically the URL stays the same up to here for each function and then the function name.

So here we have the hello world.

And then we have the get prices.

Then for adding customer card is this one.

For now these are deployed locally and we have a separate URL for each endpoint.

So in this video we learned how to implement the required function for managing subscription plans and also how to use stripe built in functions.

That's all for this video.

Thank you for watching and see you in the next one

9. Deploying Stripe Cloud Functions to Firebase

Hello everyone and welcome back.

In our last video, we learned how to create functions for subscription using stripe.

In this video, we'll be learning how to deploy back end stripe functions to Firebase and also how to test them in production with live events.

Let's get started.

So the command that Firebase gives us for deploying the function is Firebase deploy hyphen hyphen only function.

And if you want to deploy only a particular function you can go to your function's name copy it say get prices and then go back to your command prompt.

And after putting a colon at the end of this command, you can just paste it and it will only deploy that function.

So I actually want to deploy all the functions at the moment.

So I'll simply run this.

This is going to take some time.

So let's go to the documentation of the Firebase functions.

Deploy.

So here you can see the command is listed here for deploying the functions.

A deployment is almost finished here.

Let's view it from the Firebase console in Firebase I'm going to go inside the build and then the functions.

So nothing is available here yet because the functions haven't finished being deployed are now they finished being deployed.

And we will be using this URL to test it.

Because we don't have to actually pass any parameters.

So I'm just going to test this one.

This is the response.

Let's go back to the functions in the build.

As you can see these functions have been successfully deployed on our server.

And you can also copy the URL from here.

So in this video we learned how to deploy the required functions for managing subscription plans.

And after successfully deploying and testing the backend functions, we ensured secure, scalable and production ready payment processing integrated with Firebase.

That's all for this video.

Thank you for watching and see you in the next one

10. Stripe Setup

Hello everyone and welcome back.

In the last videos, we learned how to set up our backend cloud functions and publish them to Firebase.

In this video, we'll learn how to set up and configure stripe in our project.

Create a setting screen to manage subscriptions and also how to implement a plan based subscription logic with dynamic pricing.

Let's get started.

So the first thing we're going to do here is set up our stripe.

For that I'm going to go into the Android folder.

Then in our app source main and in the folder I'm going to open the main activity.

And here we'll place this Twitter activity with Fletcher Fragments activity.

The stripe payments needs code from this fragment activity to it.

Then we're going to erase folder values and styles.xml.

And here you replace this parent with theme dot material components.

Make sure the settings are correct.

I'm going to copy this from here and save it.

Also save our main activity file.

And then in the values night I'm going to replace this.

And we can just go ahead and close these files.

And in our main class you have to set up a publishable key.

I already set mine.

So you just need to go straight to publishable key and assign it your value.

And that's it.

That's all you have to do to set up stripe.

So now in my home screens folder I'm going to create a new screen called the settings screen.

Let's call this a settings screen.

I'm going to replace this with a scalpel.

So what the settings screen we're going to be getting our subscription plans.

And for that we created a function that is get prices.

And you have to access it to HTTP.

So I'm going to create a function for getting the payment plans in my payment services cards.

So I'm going to create a static method which is going to return a future of map string ends dynamic.

Get prices.

I'm going to call it.

We don't need to pass it anything.

And then I'm going to create a URI.

URI dot pass.

And I'm going to copy my URL from the firebase here.

And then just paste it.

Now that we have our URL I'm going to create a request.

This can return a response.

Await http dot get.

And then I'm just going to pass it the you are app.

And for this you have to import a library.

Import.

Package http.

And I am going to import it with an alias.

Now I'm going to check if my response says that the request was successful or not.

So the status code is if it's equal to 200, which means that it was successful.

I'm going to decode this message.

Response dot body.

Now I'm going to check if my decoded message has success.

And it's equal to true.

I'm just going to return the G codes data.

Otherwise we'll just let's say throw an exception.

Fails to fetch prices.

And I'm just going to part the G-code message here so we can know what the problem is.

Then.

Here.

That's it.

I'm going to close this file, go back into my settings.

And here I'm first going to create a list of math and strings.

Prices.

Then I'm going to create a method here that deals with the result of our response.

I'm going to call it load prices.

I'm going to get the data from the function.

So I'm going to say await.

We made it static so we can access it directly, get prizes, make it async.

Now the data is in null.

I'm just gonna set state to get the latest prizes.

So list of.

From our data.

And I'm going to create another function called format amount.

So we can get the amount from our list in integer.

So I'm going to say amount.

And return.

Which award amount is divided by 100 dot two string as fixed of two digits, and this is going to convert it to dollars.

I'm just going to call this function in the init state.

Overwrite.

Super George init state.

Override this and then just call our function road pricing.

Now in my scaffold I'm going to create the app bar.

Set its title to.

Text of payment options.

Style.

I'm going to set the text styles color to color stored white.

And then I'm gonna say center title is true.

And I'm going to set the background color to our app's theme.

So color dot orange make this constant.

After that I'm going to set the body.

Let's create a list view because I'm just basically going to be using tiles.

Children, let's create a list tile.

And set its leading icon to icons dot log out.

Because this is where we're moving our logout logic to.

And then I'm going to set the color to our theme.

Title is going to be text Log out.

And on tap just go ahead and use auth services.

Dot sign up.

After this trial I'm going to create a divider and set its color to orange.

Then I'm going to use theme to set the entire theme of this.

I'm going to set it to Expansion Tile and its title is going to be.

Subscription.

Inside theme I'm going to set the data theme Seemed out of context to copy with.

The wide color.

Palette.

Dodge points.

Like this.

Constant.

Sets the leading icon here.

I'm going to say leading.

Icon.

So it's credit card.

Sharp with the color colors to orange.

Now I'm going to set its children.

And for that I'm going to use my prices map.

So I'm going to say prices dot map.

I'm going to set this to price.

And it's going to return a list.

I'll.

Move on from here.

I'm going to create a new function here called string capitalize.

String of some text.

If this text is empty, return text.

Otherwise return text or 0.2 uppercase plus text dot substring.

I'm going to call it here.

Finally going to get the nickname.

Rice.

Nickname.

Otherwise just some random string, say unnamed.

Clan.

Make this two list first.

After exiting the name, I'm going to also access the amount.

Which is price?

Unit amount.

Otherwise, set it to zero if there is nothing in there.

Now inside my list I am going to set the title to text.

Nickname.

With whichever the amount is.

So format amount function that we created and pass it amount after title set the trailing icon.

So I'm going to say icon.

Icon store.

Arrow.

Forward.

And set its color to colors dot orange.

Style our text also.

So style textile.

Color.

Orange.

Dodge.

Shades of 900.

Now after setting the trailing, I'm going to go ahead and work on my own tab.

And here we're gonna enter our stripe logic.

And then I'm going to set the tile color to color orange dot shades 50.

Let's run it.

So let's go into settings.

As you can see our payments option view is here.

And it's fetching these early and monthly from our backend endpoint.

So in this video we learned how to set up stripe in our project and also how to enable subscription plans handling using backend driven plans IDs.

That's all for this video.

Thank you for watching and see you in the next one

11. Implementing Payment Using Stripe

Hello everyone and welcome back.

In the last video, we learned how to create a setting screen to show our subscription plans.

In this video, we'll be learning how to integrate a payment dialog into the project to enable secure processing directly within the users interface, and also how to ensure that the user can successfully and safely input their card details.

Let's get started.

So I'm going to go into my payment folder and create a new file here called the stripe dialog, same as our PayPal dialog.

Make it a stateful class.

I'm going to name it Stripe Details screen.

Replace this with a scaffold.

Above.

I'm going to set its title to nothing, because I'm going to be setting a heading below in the body.

So let's just say it's nothing.

So the app bar still shows.

Some padding.

I'm going to set the padding to edge insets George only.

Left is going to be 15.

Right is going to be 15.

Top is five and bottom is also five.

After padding I'm going to make a child of it called the Single child.

Scroll here.

And set each child to a column.

Children.

Set its main axis size to main axis size.

Roadmap.

And the cross axis alignment to axis alignment dot start.

I'm going to create a size box to create some space from our app.

Then I'm going to create a center widget that's going to create the heading.

I was talking about so text is gonna be center stage or we're going to call details.

Then I'm going to set its text align to center.

Also.

Style.

Text style.

Font size is 20 and font weight is binary font weight about.

Then after center, I'm going to create another size box, which I'm going to set to 25.

Now let's create the header before our text form field.

So text is going to be name.

Text align.

Start.

Style.

Textile font rate is W 500 and font size is 14.

After the sidewalk.

So this is going to be five.

Then I'm going to create the text font page.

Set a controller for this.

So for controller I'm going to create them in my Payment services class so I can access them easily.

And I don't have to pass data between the screens.

So I'm going to make it a static one.

I'm going to name it card number.

Ctrl and then text editing controller.

Copy this.

I'm going to need three more.

One is going to be date.

CVC and of course the name.

Save this and uncomment this.

So I'm going to say payment service dot name controller.

Then I'm going to set its decoration.

So const input.

Decoration border.

Outline.

Input border.

I'm going to enable border outline input border and set the color for the border side.

So color is colors dot gray.

Then we have our border.

Or when we click on our text form field and border side color.

Palette store orange.

Now I'm going to set its width a little bigger so it's more prominent.

And paste it.

This one is going to be ten card number.

And inside of this we're going to have card number controller.

Set its keyboard type to say text input type.

Number.

And here I'm going to set the input formatter.

23.

Text input format or digits only.

And I'm going to limit its length.

To 16 because the maximum number is 16.

And here I'm also going to set some label text so the user knows what format they're supposed to enter the text chain.

Now I can just go ahead and copy this again.

Now I'm going to right on the expiry going to limit its length to 15.

Set this to date time.

And this is going to be something some example dates right.

Then the next one.

CVC.

Only three.

And I'm going to set the max line for this to one.

Any three digits.

Now that we finished checking our inputs I'm going to create a size box.

A little bigger height is going to be 40 then the elevated button.

Some text is submit.

I'm going to set the style for this elevated button.

Style from padding.

As insets.

Symmetric.

Vertical is 16 while horizontal is 140.

Trying to set the shape to rounded rectangle border.

Set the border radius.

Circular.

I'm going to set it to 12.

Then I just have to set the background color and the foreground.

The background color is going to be orange.

And the foreground color is just going to be colors dot white.

Save this file.

So we also need to change the controller here.

So it's.

CVC.

And then we have the state controller here.

Save this file.

I think that's about it here.

And we also need to go into our home screen and settings screen.

And here we need to replace the stripe logic.

Navigate to the class that we created for stripe.

So navigate to dot context dot push material page route context.

Stripe.

Details screen.

Yeah, I think that's about it.

That's all about we need to change here.

Colors and everything.

Let's run it.

So let's go into settings subscription and clicking on either of these I'm going to click on monthly.

As you can see we can see our entire details screen with all our information.

So in this video we learned how to establish a secure and user friendly method for handling payments directly within our application, and also how to import card information directly in the users interface.

That's all for this video.

Thank you for watching and see you in the next one.

12. Tokenizing and Transferring Card Details

Hello everyone and welcome back.

In the last video, we learned how to create a stripe dialog for storing card information.

In this video, we'll learn how to use back end endpoint to securely create and store a card object, and also how to modify the stripe payment dialog to transmit card details to the back end endpoint for processing.

Let's get started.

So to create a card using my endpoint I'm going to go into my Payment Services class.

And here I'm going to create a new method.

Future bool.

And I'm going to call it create card method.

We have to pass age the build context.

Async I'm going to provide a try catch block here.

Then I'm going to create a card token parameter.

I'm going to call it called params equal to const.

Let's import this first stripe library.

Then I'm going to import the card token parameters.

And I'm going to pass it the type which is token type dot card.

And I'm going to say await stripe dot instance dot dangerously update card details.

The card details object.

I'm going to pass it the number which is going to be the card number controller or text.

Then it's going to be the CVC.

So CVC controller the text.

Expiration months I'm going to pass it to interface because it's going to be a string.

So date controller, dot text dot split on the basis of the slash and it's zeroth index.

Copy this.

We have the expiration month.

And for the expiration year.

I'm just going to turn it to one.

And that's how you create the card.

Now I'm going to create a token which is the most important part in this drive.

The token is what we're going to be sending to our backend endpoint.

So stripe dot instance dot create token I'm going to pass it the create token.

Parameters card.

And I'm just going to pass it the card params.

Now I'm going to create the URL.

You are right, good parts and we just pass it from the firebase.

This one create card.

Paste this here.

Then I'm going to create the header.

So.

Content.

And then application slash JSON.

Then I'm going to get the email.

For that I'm going to have to go into my auth screen auth services.

And here I'm going to create a new method for fetching the email.

I'm going to say static string.

Get current user email.

Create a user.

Third instance.

Dot.

Current user.

And then I'm just going to go ahead and return its email.

Save it.

Now I'm just going to access the email through my auth services dot get current user email.

Then I'm going to send the request which is going to return a response http post.

I have to pass it.

The URL headers is going to be the header we created.

Body is going to be JSON encode.

Card token is the first thing we have to pass it.

And we're just going to pass the token ID.

Then we have the name.

So name controller dot text.

Then there's the email.

So email.

And if it's empty don't pass it anything.

So I'm going to go ahead and decode the JSON response.

I'm going to pass a response for J.

And then if response dot status code is equal equal to 200 then that means it was successful.

I'm going to check if the data of success exists.

Then I'm going to go ahead here I'm going to be saving my storing customer ID.

Function.

But for now, let's just navigate.

So I'm going to pass the material page route.

Context.

And I'm just going to take it to the home screen.

Since adding the card was successful.

And predicate is going to be a route.

That is dynamic.

Route is going to be false.

I'll shift F I'm just going to go ahead and show a message here.

Of context.

Don't show.

Snack bar.

Snack bar.

It's gonna be.

Let's just say.

You're now a premium.

Members.

Way to go.

Make this constant.

And I'm just gonna return true from here.

And now, if that wasn't successful, I'm just gonna copy this message and return statement.

Change this to false.

And I'm just gonna show here.

Payment fails.

And if it's empty, just show transaction.

Where?

You move this const.

In here.

I'm also going to be doing the same thing.

Let's just say we were unable to save your card details.

This is also going to be false.

As you say something went wrong.

Falls from here also.

Save this.

I'm going to go into my stripe dialogue and on my own tab of this submit button I'm going to check is card Save.

And I'm going to await payment Services dot create card in this pass in the context.

And I'm going to check if the result is true then make it false.

Otherwise true.

And if it's not, just return it.

If the transaction failed.

Let's go ahead and run this.

Let's go to settings.

Subscriptions and just choose one of the plans here.

Then I'm gonna input my details, let's say John Doe.

So these are the sample card that's provided to us by stripe.

And you can use this information.

So after the card number which has to be exactly 16 digits I'm gonna enter the expiry.

Let's just say 2426.

And then the CVC which can be any three random letters for now against this card number which will always

be used as testing.

As you can see, we were able to save our card details.

So in this video we learned how to securely handle card creation and storage on the backend, reducing

exposure to sensitive data and improving compliance.

The payment dialog is ensuring a smooth and secure flow of card information.

That's all for this video.

Thank you for watching and see you in the next one.

13. Generating Customer Stripe ID

Hello everyone and welcome back.

In the last video, we learned how to create a card in stripe using our backend endpoints.

In this video, we'll learn how to automatically generate a stripe customer ID upon successful card

creation, and also how to retrieve customer stripe it seamlessly from Firebase for payment processing.

Let's get started.

Last time we left our create card function a bit unfinished, where we had to create a function to store

the customer ID.

So let's create that function now.

I'm going to call it Store stripe customer id.

I am going to be passing it a map of string and dynamic, which will basically be the response to our

backend API.

So the API response.

Points.

Going to make it async.

Going to wrap it in a try catch block some boilerplate code.

Now I'm going to create the customer ID that is customer ID from the API response.

So data.

Check customer ID and if it exists take its ID.

Now if customer ID is not equal to null.

Just throw an exception.

Customer ID not found in API response.

And in the case that it does exist, I'm going to get the user From my firebase auth.

To instance import this.

Current user.

If the user is equal equal to null.

For another exception.

User is not logged in.

Then this will be a rare case because user will always be logged in.

User is equal to firebase.

Fire storm dot instance.

Dot collection.

User dot ui.

Now I'm going to say await user dot dot set.

The data is that I want to search here.

I'm going to name the field stripe customer ID and then just pass it the customer ID.

In case the exception is luggage stripe customer ID.

And that's it.

I'm just going to call this function in my card here.

Pass it the data.

I'm also going to create a function to fetch this ID so we can use it when we're creating our payment request.

So I'm going to create a static.

Picture of string.

It's just going to be returning the ID.

So it's stripe customer ID.

Left.

Wrap it in a try.

Catch.

Just log it.

As.

Stripe customer ID.

From here, I'm just going to return null because we won't have the ID here.

Then I'm going to retrieve the user again from Firebase auth.

Dot instance.

Current user.

And then I'm going to get the user doc.

So I just say await Firebase Firestore dot instance dot Dodge Collection.

Of fuses.

Dot.

Dot.

User.

Dot.

Dot.

Get.

And then I'm just going to get the customer ID from stock.

So I'm going to say user dot dot data.

If it's not null that means the stripe customer ID which is this one right.

So there are no spelling errors.

And then we just return the customer ID from here.

I'll shift F.

That's it.

That's how you store the customer ID and save it from our API response.

So another thing I'm going to be changing here is I'm going to be adding a price variable here, which is going to be a string.

It's going to call it price.

And I'm going to be fetching the price from where we have our payment plan on tap.

I'm going to go ahead and access it through payment services.

Price is equal to price of ID because we're not actually going to be passing the price amount to our

Firebase to our backend function when we're making the payment, but rather the ID.

So in this video we learned how to create a stripe ID for each user to uniquely identify them for transactions,

and also how to centralize stripe customer IDs in Firebase, ensuring consistent and reliable access

during payment operations.

That's all for this video.

Thank you for watching and see you in the next one.

14. Securely Handle Subscription Creation with Full Error Handling

Hello everyone and welcome back.

In the last video, we learned how to create a customer stripe ID during card creation.

In this video, we'll learn how to handle subscription creation securely via a backend endpoint to maintain

and manage control and data integrity.

Also, we'll learn how to implement comprehensive error handling and cover all path executions during

the payment process.

Let's get started.

So after create card function I'm going to create the create payment function.

To create payment you can name it create subscription.

It's the same thing either way.

So we're going to be passing it the context and a string.

Wrap it in a try catch block.

So here I'm going to be accessing the URL to create the payment.

Go to Firebase and this is our subscribe method.

So copy this.

Then I'm going to create the headers.

Content type is.

Application of JSON.

And I'm going to be accessing the stripe customer ID.

So the function we created in the last video I'm going to say fetch stripe customer ID.

Now I'm finally going to create the request of post.

So I'm going to say http dot post twice.

We pass it the URL.

Then there are the hedges.

And lastly we have the body Where we JSON encoded first and then send it a map.

Testing is going to be customer ID because that's what the function requires.

So I'm just going to pass it the stripe customer ID.

And then there's the price ID which we also get from our settings screen.

So price.

Now I'm just going to decode the JSON response.

And if response dot statuscode is equal equal to 200 I'm going to check if the data has a success in it.

And if it does I'm just going to get the subscription ID.

From data.

and its ID as a string, which may be.

Then I'm going to create a function here to save the id and return true.

Otherwise just.

Otherwise just return false.

And here.

Here I'm going to get the failure code from data as message.

Since if we get to this point, it means our payment wasn't successful.

So I'm going to get the failure message also which can be declined by default.

And if failure code contains.

Insufficient Thanks.

Again for your message is going to be.

Your card has been declined.

I'm just going to copy this.

And I'll say it says.

Incorrect CVC.

These are the most common scenarios that can occur when making a payments.

That's why we're handling them.

Please check your card's details.

And if this is your card has expired.

Just say your card has expired.

Else here.

Lastly, I'm going to show a snack bar.

With that failure message.

Notice.

And I'm just going to return false from here.

On I'm just going to return false.

And there is another kind of exception here on stripe exception.

Catch it.

This can be really useful for showing a specific message if it was a failure due to stripe.

Each error message.

And if that's empty just show.

Payments fails.

Please try again.

Remove this.

It's also we're going to return false from here.

Now I'm going to go ahead and call this in my stripe dialogue here.

Await payment services dot create payment and just pass it the context.

Let's run it.

So go to Settings Subscriptions and I'm going to click on the early one.

Let's input the fields that we did before.

The expiry in the CBC submitted.

And there we go.

We have now shown our message.

And I realize now that it's the same as the last time, because you can't actually see any difference.

But the payment was created, and we're going to show in a later video how it affects our polls, because we were only allowed to create three before.

If you are a premium member and now you can create more than three if you're a premium one.

So in this video, we learned how to achieve secure subscription payment handling through controlled backend endpoints, and also how to do thorough error handling to create a smooth and trustworthy payment

experience.

That's all for this video.

Thank you for watching and see you in the next one.

15. Managing Subscription Data

Hello everyone and welcome back.

In the last module, we learned how to create subscription payments from user.

In this video, we'll learn how to store user subscription data in Firebase after successful subscription.

And also how to retrieve and manage subscription records.

Let's get started.

So now I'm going to go ahead and create this function for saving the subscription ID.

Provides save subscription ID.

And we're going to be passing it the subscription ID.

Make it async.

And I'm going to wrap it in a try catch block.

Firebase auth dot instance dot Write and create the catch here.

Just going to go ahead and print the exception here to save subscription.

I am going to check here.

If the user is equal equal to null.

Then I'm just going to throw an exception.

Now user log.

Then I'm just going to add it to the Firebase Firestore.

Collection of users dot doc and user dot UI.

And I'm going to update it With the following map, which is subscription ID against the subscription

ID we receive.

A shift F and that's it.

That's all you have to do to save it.

Next I'm also going to create a retrieve function for it for when we need to retrieve it from our endpoint.

It's going to be returning a map of string.

And dynamic.

So I'm just going to fetch user.

Subscription.

Make it async.

Final user is equal to.

Firebase auth dot.

Instance dot current user.

And then I'm just going to retrieve the document.

So await Firebase dot instance of collection of users.

User dot doc uid.

And then I'm just going to get it.

And I'm going to check if it's not.

Exist.

Return null.

Then I'm going to get the subscription ID from it.

So subscription ID is equal to user doc dot data.

If it's not null then get the subscription ID.

As a string.

Next I'm going to check if this subscription ID.

Is equal equal to null, or if this subscription ID is empty.

In both cases, we're also going to be returning null from here.

And this will fetch our user subscription from Firebase.

Now I'm going to call my subscription ID here.

Then just pass in the subscription ID.

And make this nullable.

Save this file.

Let's go ahead and run it.

Let's go into settings in the subscriptions.

I'm going to create a new subscription here.

So any random future values I'm just going to do this one.

And CVC submit this.

So let's go into Firebase and check if our subscription ID was saved in the users.

As you can see it was saved successfully.

And along with our stripe customer ID.

So in this video we learned how to use Firebase to store and manage subscription data, and also how

to use Firebase queries to efficiently retrieve user subscription records.

That's all for this video.

Thank you for watching and see you in the next one.

16. Implementing restrictions in Free Tier

Hello everyone and welcome back.

In the last video, we worked on handling subscription during the payment process.

In this video, we'll learn how to implement a usage restriction that allows users on the free tier

to create up to three polls encouraging upgrades for extended functionality.

So here I'm going to create my function to fetch my subscription data from my backend endpoint is going

to be returning a string of map.

And I'm going to say retrieve subscription.

We're going to be passing it a string subscription ID.

Make it async.

Now I'm going to get the URI from the URL.

So you are I dot parse.

And I'm going to copy the link from from the Firebase functions, the retrieve subscription one.

And I'm just going to paste it here.

Now that we have the URL I'm going to create a request to it a Get request.

So response is equal to http dot.

Get this URL.

And if the response dot status code is equal equal to 200.

Then I'm going to get the string dynamic data from it.

Which means theta is equal to JSON decoded from our response dot body.

Then I'm going to check if the response data has success in it and is equal equal to true.

In which case I'm just going to return the response data and the data field in it as a map of string

and dynamic.

Otherwise, I'm just going to throw an exception.

Fail to retrieve subscription.

And I'm just going to show whatever the message is in my response data.

And if the response status code is not 200 I'm just going to throw another exception here.

HTTP error means.

And instead of this I'm just going to show the response dot status code.

Now I'm going to call this function in my fetch user subscription.

Wrap it in a try catch.

Final subscription data is equal to await retrieve subscription and just pass in the subscription ID

and just return subscription data from here.

And in the catch block.

I'm just going to log the error.

Retrieving.

Subscription.

And just return null from here.

Save this file.

And then I'm going to go into my home screen and show polls.

And in my builds.

Before navigating to the create screen.

I'm just going to check if user is subscribed or not.

And in case they aren't subscribed they are only allowed to create three polls.

So polls software.

Creator id.

Is equal to.

Current user ID and then I'm just going to Retrieve that.

Make it async.

Then I'm going to get my map.

From my function that we just created so it subscribes is equal to await payments services dot fetch

subscription.

Now I'm going to check if the snapshot dot dot dot length is greater than or equal to three and is subscribe.

Is equal equal to null because it won't exist.

If the user is not subscribed, then we can just go ahead and show a message here.

On the show.

Snack bar.

Snack bar and show text here.

You can only post up to three posts on the three.

Here.

You can upgrade your plan in settings.

And I'm just going to return it from here instead of letting it create a new poll.

Save this.

So another thing we need to change here is the URL to our endpoint.

We're going to be passing it the subscription ID also.

So after a question mark the required parameter that's subscription ID and just ask that.

And then you can just save the file and that's it.

For now I'm going to go into Firebase and delete the subscription ID because here we're also checking if it is equal equal to null.

And let's make it null first to see if our logic is working correctly.

Delete it.

Let's run our project.

So in our posted section We already have these three poles created by this user.

I'm going to try and add a new one.

As you can see, it's all showing.

You can only post up to three poles.

Let's go into settings and create a subscription.

And submit it.

So it says we were subscribed.

Let's try and create a new pole here.

Now.

As you can see, it's allowing us to create a new pole.

In this video we learned how to enforce tier based limitation to differentiate user capabilities and drive conversion to premium plans.

That's all for this video.

Thank you for watching and see you in the next one.

11_project-branding-and-documentation

1. Creating a Professional README file

Hello everyone and welcome back.

In the last video we finished our stripe functionality.

In this video, we'll learn how to create a Readme file for providing guidelines about the feature and other description of the project, and also how to use markdown elements like headings, lists, and horizontal lines with proper indentation.

Let's get started.

As you can see, I've already created a readme file here, so an important part of Readme file is how it looks.

So you need to know the proper indentations.

So the double hash here is going to create a heading basically like a heading two in Ms-word.

The hash is going to create sort of a bigger heading is both of these are going to be bold and larger font size.

So first you provide the name of the app.

Then you provide a description that describes what your project actually does.

After that you just provide the features.

What it is your app provides that makes the user want to download it.

So for example, we can allow users to create and waterfalls and location based discovery.

We also for offline sports and then we also for monetization.

Then this is the most important part of a Readme file.

You need to provide a setup guide.

For example the filter version.

This is also very important.

Here I've used the filter version 3.16.2.

So if you're not on the latest version of flutter in your project, you definitely need to mention that here.

And then I've also just said that these accounts and API keys will also be required to run this project.

Then you just provide the link to your GitHub repository.

I've provided the link to mine and just create git clone and just put the link and it will go on it.

And then these three and five are not necessarily required.

But you should add these because if a beginner um student or someone new to run the project, they won't

know these kind of things.

And then you provide the API keys we've used Firebase, Google Maps and Stripe and PayPal.

All of these require API keys, and we also tell them that they need to make changes in all of these

folders.

Android and iOS.

And for Google Maps, you're going to need to make changes to these two folders.

And for stripe, they're going to be creating a back end of their own.

Then we provide the usage of how our app works.

You can just create a poll and discover them both on them.

And then we've also established the offline mode and then managing subscriptions.

Another important part is the tech stack.

You also need to provide what you're using for state management for example provider here.

And then there's also get access block.

So you just mention that here.

And you also need to mention that we use Firebase SQLite and Sharedpreferences for storage and Google

Maps and Stripe and SDKs for different reasons.

For this one we use it for location and this one for monetization.

And the last important thing is the license.

You need to mention what license you are using for this project.

MIT license basically means that it's an open source project, so anyone can make request and push code

to it and basically just make any sort of contribution they want.

If you want a custom license that provides a bit more restriction, you can create that and that's it.

That's all you need to do to create a README.md file.

I've already pushed it to GitHub.

As you can see here, it's providing us with all that we've added to the readme file here.

So to make sure that the readme file shows here, all you have to do is push it to GitHub and it will

automatically show up here in this section.

In this video, we learned how to apply consistent formatting and structure to enhance the readability

of the Readme file, and also demonstrated effective use of markdown to deliver clear, organized, and developer friendly documentation.

That's all for this video.

Thank you for watching and see you in the next one

2. Generating Custom App Icons

Hello everyone and welcome back.

In the last video we learned how to create a Readme file.

In this video, we'll learn how to generate platform specific app icons optimized for Android and iOS.

In all required resolutions and also how to add icons to respective folder in project so the icon shows when the app is installed.

Let's get started.

So to generate icon folders that we need we're going to go to the app icon generator.

So I'm going to go into the app icon generator.

And here it's going to give you options for all this iPad watch.

I'm just going to choose iOS and Android for now and just uncheck all of these.

Then you provide it your icon.

It can be any image file.

So I'm providing it with this icon that I've already generated.

And then I'm just going to click on generate.

And there it's going to give you an entire folder with all of this image for App Store and Play Store and all the resolutions required for Android.

All the different resolutions required for iOS.

So what you're going to do with this is go into the Android folder, and then you're just going to copy this.

And then I'm going to go into my project and I'm going to open it in the File Explorer.

So I'm going to go into Android App source main.

Mean and then resolution.

And here I'm just going to go ahead and paste this control V.

And then I'm going to say replace the files that are the same.

And that's it.

So now I'm also going to go ahead into my.

Just going to unzip this first extract all.

Now I'm going to go into the assets one.

And here I'm going to just go ahead and copy all of this also.

And then I'm going to go back into my iOS folder.

Open it in finder.

And then in runner assets, dot x assets and app icon dot app icon set.

Once you select all of this and delete it first.

So delete.

Then I'm going to paste it here.

And another thing I want to paste here.

Another file that comes with this this content one.

So I'm going to copy this.

And then I'm just going to go back.

So here I'm back in my iOS.

And I'm just going to paste this here.

So first you going to want to clean your project.

So to Fletcher clean.

And then I'm just going to run Fletcher Pubget.

That should finish up.

So I'm just going to go ahead and run it now using dart.

Let's check the icon.

And as you can see it's showing the right updated icon instead of the default Fletcher one.

So in this video we learned how to implement a platform specific asset pipeline enabling consistent icon rendering and branding across iOS and Android applications.

That's all for this video.

Thank you for watching and see you in the next one.

3. Flutter Chatbot UI Design – Building the Chat Screen (Part 1)

Welcome to this module on building an AI chatbot in Flutter.

AI-powered chatbots are becoming a common feature in modern applications

because they allow users to interact with apps using natural language

and receiving intelligent responses in real-time.

In this module, we'll explore what AI chatbots are,
why they're valuable for mobile applications,
and how we can integrate one into our Flutter app.
So, what is an AI chatbot?
An AI chatbot is a conversational interface
that allows users to communicate with an application using natural language
just as they would with another person.
Modern chatbots are powered by advanced AI models
such as Gemini and ChatGPT,
which can understand context,
generate human-like responses,
and assist users with a variety of tasks.
So, why add a chatbot to your Flutter app?
AI chatbots provide several benefits for both users and app developers.
They can offer instant assistance and support
without requiring users to search through documentations or help pages.
Chatbots also increase users' engagement
by creating a more interactive experience
and encouraging users to spend more time in the app.
So, let's get started on building a chatbot.
So, let's start off by creating the UI for our chatbot.
For that, I'm going to head into my Features folder
and create a subfolder, the Chat Screens.
And then inside it, I'm going to create a chatscreen.hack.
Make it a stateful widget.
The chat screen.
Now, let's replace our placeholder.
I'm going to start by setting my app barcode.
Let's remove this.
And I'm going to set its background color
to our color scheme that we already have.
So, color scheme.primary.

And let's import that.

Final. Color scheme

is our theme of color scheme.

Let's rename this.

And then I'm going to set its title spacing to zero.

Let's start by setting up our title.

ShowCircleAvatar, which is basically going to be our AI.

And replace this by Arto Awesome to represent our AI.

And then I'm going to set its size to 18.

Let's try finding this. Arto Awesome.

Then below this, I'm going to add a size box.

The size box with a width of 10.

And then I'm going to create a column.

I'm going to first set its cross-axis alignment to start.

And then a text.

Which is going to be AI Assistant.

Set the right style.

And then text style.

I'm going to change this to 18.

And then we want to set it to W600.

Color is going to be white by default, so let's just remove that.

Next, let's go ahead and create a provider for our chat screen,

so we can work on it simultaneously.

So, a new file in our providers.

I'm going to call it the chat provider.

And let's create chat provider, which extends our change notify.

I'm going to create a is loading variable here.

And then also a getter for this.

So, bool get is loading.

Now, let's head back into our app bar.

Now, we're going to consume that here.

So, add a consumer.

And let's provide its name.

That's going to be the chat provider. Import it.

Let's rename this to our provider.

And then depending on our state, we're going to be showing the status.

So, if provider.isLoading, show typing or online.

Set its font size to say 12 instead.

And then according to our state, we're going to check provider is loading.

If so, then just use the primary color scheme. Remove this.

Otherwise, just show that you can type.

So, let's show it green.

And let's add a shade of 600.

So, our app bar is all set.

Now, let's move on to the board gig.

I'm going to replace this with a column.

And then use an expanded widget.

To each child, I'm going to provide another consumer.

So, let's use the chat provider again.

Next, we're going to head into our chat provider.

And create a new list for our chat messages.

This is basically what we will be showing to the user.

So, we don't have this yet.

So, next I'm going to go into my models folder and create a new chat model.

And I'm going to create an enum here called the message sender.

Which will basically tell us whether it's the user or the AI that's sending the message.

Then I'm going to create my module class of chat message.

First thing we'll have is our dog ID. From Firebase.

Then we'll have our string. Text.

And our message sender.

Which will basically be the sender.

And then the timestamp.

Let me quickly create the tool map.

Since we're using the chat provider,

Since we've already gone through the basics of this, I don't need to go very in-depth for it.

So, chat message.

So, our model is set up.

Let's go back to our chat provider and import it.

Next, let's create a getter for our messages list.

So, our list of chat messages.

And let's make it an unmodifiable list.

Because the messages are going to be unmodified.

And let's make it an unmodifiable list.

Because the messages are not going to be changed.

The previous ones.

So, let's just pass it the message.

So, now that we have our messages, let's go back to our chat screen and continue building on that.

Now here, the first thing we're going to check is if our messages are empty.

So, if our provider.messages.isEmpty.

I'm going to return an empty state from here.

And let's pass it our color scheme.

So, our class of empty state.

Let's make this a stateless widget.

Set up the padding first.

And to each child, I'm going to assign a column.

And then the first thing I'm going to use here is an icon.

Which is going to be icons.Auto. Awesome.

And I'm going to set its size to 64.

And the primary is going to be with values.

Let's say alpha.

And 0.3 should do.

Something helpful.

How can I help you?

And then let's set the style.

And then I'm going to use another text here.

Us, me, anything.

So, now the empty state is set up.

Let's go back to our main screen.

And let's show our chat screen.

So, now the empty state is set up.

Let's go back to our main screen.

And let's review what errors we have.

Now, from here, since we're returning an empty state in the if block, we also need to return something else.

Let's set the item count.

So, it's going to be our message's length.

And then if our provider is loading.

Let's set it to 1 or 0.

I'm going to set its padding.

And then I'm going to set its height.

I'm going to set its padding.

And from here, we're going to be returning.

So, return.

A message bubble widget that we're going to create in a moment.

And to that, we're going to be passing a parameter message.

And the message we have.

Although if the provider is still loading here.

Is loading.

And.

If our index.

Is equal to provider message length.

I'm going to return a typing indicator.

Which is also something we're going to create in a moment.

Which is basically going to show that AI is currently typing.

So, let's go ahead and create a topic indicator next.

Below our empty state, I'm going to create a stateful widget.

The typing indicator widget.

I'm going to start by setting its.

I'm going to start by setting its.

Build to padding.

And insert the padding.

I didn't set start only.

And I'm going to set the bottom to 12.

Close this.

And then since it's going to be sort of an animation.

I'm going to extend it with mixing.

The single jigger.

Which is basically used. For animations.

As you can see to create the animations.

And then I'm also going to have to create. A.

Animation controller here. Animation. Controller.

And then set up my innate state, which is going to initialize my controller.

And I'm going to set it to 900. Duration.

After 900 milliseconds, it's going to restart the animation.

We also need to make sure to dispose it.

So there are no leaves. So overrides.

And then dispose it.

In the build, I'm also going to set my color scheme so I can use it later.

First is going to be a circle avatar, which is basically going to denote our AI.

Instead of this animation builder, I'm going to use a container.

So let's remove this.

Add a container here.

And symmetry.

I'm going to set the horizontal to 16 and then the vertical to 12.

And then once we're done with our decoration, I'm going to assign it a child, which is basically going to be in animations.

And I'm going to assign it.

A child.

And then I'm also going to assign it a child. And. Animated.

Builder, which is going to build our animation.

I'm going to assign it the controller we created.

And then to the builder.

We don't need either of these, so let's just skip those. And.

Let's assign it a role.

To the children, I'm just going to directly assign a list to generate.

And it's length.

It's going to be three.

Just three doors that we need.

To show that AI is loading our response. Our response.

And then we're going to set our animation delays.

So final delay. Is going to be.

I. By three.

Then T. Then T.

I'm going to assign it controller.

Dot value minus the delay.

And we're going to clamp it.

0.0 and.

1.0

Finally, I'm going to set the opacity.

You're going to see in a moment how we're using the T variable we created. 0.3 plus 0.7. And.

This is basically raw code you can put full of the Internet.

Now, let's use it. We've done.

A padding with it from here.

Padding.

So now our typing indicator is ready.

So in this video, we explored the fundamental of AI chat boards.

Why we need them basics on how they work.

And finally, how they help scale our application.

We'll continue building our chat screen in the next part.

Thank you for watching and see you in the next one.

4. Flutter Chatbot UI Design – Building the Chat Screen (Part 2)

Hello everyone and welcome back. In the last video, we were working on our chatbot UI and in this video, we're going to continue on that. So let's get started.

As you can see, let's close this. Our typing indicator is working. Let's create our message bubble next. Let's make it a stateless widget because it's a simple bubble. User doesn't interact with it. I'm going to call it the message bubble. Let's remove this and I'm going to be passing it our message bubble. The chat message we created. Let's import that here. Okay, so next I'm going to go and replace the placeholder again with our padding widget. So first we're going to check if it is the user or the AI that send the message.

So message sender. Next, we're going to check if it has text or not. So message.txt is not empty. Now let's get started on our message bubble. Let's set the cross axis alignment to the end and then the main axis alignment is going to depend on whether if it is the user or not. So user. If it is, then it's going to be on the right side. Otherwise it's going to be on the left side. So. Now if it is the user, then we're going to show our avatar. So it's going to be circle avatar. And I'm going to set the radius to 16. Next, let's use a flexible widget so we don't have any overflows. And I'm going to assign it a column. And then let's say the columns cross axis alignment depending on if it's the user or not. So if it is the user, it should cross axis alignment or end. Otherwise, this stuff.

And let's set the children. I'm going to use a container for the bubble. I'm going to set each declaration to box declaration. Color is going to depend on whether it's the user or not again. So you can see how our enum is coming in handy now. Because we constantly need to compare whether it is the user or the AI in our chat screen.

Now, if the text is present, only then show the bubble correctly. So padding. I'm going to set it to 14. Now let's move on to each child. Assign it a text, the text of the message, the message text. And then set its style. I'm going to set the font size to 15 and the height is going to be 1.4. And then below our text, we're going to show our timestamp for when the message was sent. So let's use a size box first. Height is going to be 4. And then the text I'm going to use the date that we get from our message. So date format. Let's import Intel.

And I'm going to set the format to time. And then format our message to a timestamp. Let's set its style. So text style. Font size is going to be 11. And then the color scheme. Font surface with values of alpha, zero point. That's a profile. We set our UI for the AI. Let's check if it's the user. In which case, we're just going to show a size box. And then also the circle avatar. I'm going to set

the radius to 16. Background color is going to be color scheme. Secondary. Now we're also done with

our message bubble. And this is all set. Now we want that whenever a new chat opens or whenever a

new message is sent, that the chat is scrolled down to the bottom automatically. So for that, we're going to

go inside a list view builder. And then I'm going to see here the widget bindings. After the widget is done

building. It's going to be scroll to bottom. Now let's move above our build. And let's create a method here.

Widgets scroll to bottom. Inside this, I'm also going to check to make sure we do not have any errors.

The widget findings.instance. Post frame callback after the widget is done building. Now I'm going to create a

scroll controller. So scroll controller, which is aware of the state of the screen. So scroll controller. And

then here, I'm going to check if the scroll controller. Has clients. Has clients basically tells the scroll

controller if the screen was scrolled automatically. Or by the user. So scroll controller.AnimateTo. And then I'm

going to send the offset to.position.maxExtent. Go to the bottom of the screen. Duration, I'm going to set to

milliseconds.300. And then finally, the current is going to be. Now we have our scroll controller

set also. Finally, we

need an input bar on our screen. So user can type a message. For that, I'm going to go below my list view builder in

outside the expanded. So it will set the bottom. I'm going to create an input bar. To that I'm going to pass the

controller. For controller. I'm going to set a function. On send. Printing image, you're going to see how this

printing image comes in hand later when we insert images also along with text. For now, it's just to send message. And

then context.read. I check provider. I'm going to create those. First, I'm going to create the send message.

It's going to be needing a chat provider, right? As we said, so provider. And the final text is

going to be controller.text.trim.

Let's create design the controller. The controller is going to be a text editing controller because

it's a text box. If our text is

empty, don't do anything. Just return it from here. Otherwise, controller.clear. And then here we're going to do our

create message function in our provider. We haven't created that yet. So let's save this. And afterward, it's going to

scroll to bottom. Now our input bar is pending. So let's go ahead and create that next. I'm going to quickly work through

input bar. Let's make it a stateful widget. And we're going to need two parameters here that are required. So this dot

controller and required. Let's create a text editing controller here. And then I'm going to create

the function. For now, let's make

it an X file for later when we integrate our image. And then the on send function. Let's import

this. I'm going to create a handle

send function. Widget.onsend since you don't want to call it directly. For now, let's set it to null

because we won't be sending an image.

Let's start on our input bar. I'm going to need a color scheme here. So let's create a variable

called that. Replace this with a safe area

widget so it's not hidden behind the navigation buttons at the bottom of most Android phones. Assign

it a container. And I'm going to set

the padding. Create a child in our container quickly. Assign it a column. And set the main axis size

to main axis size

to a minimum. And then the children. So for the input bar, I'm going to use a row and going to use

an expanded inlet. Assign the child a text

field to the controller. We've already initialized a controller. So widget.controller. And then let

me quickly set its properties. On submit it.

The function we've already created. And then let me just quickly set the decorations.

Okay, now we're done with our input bar. And then I'm going to create a send button. So for that,

we're going to need a consumer or the chat provider. So I'm going to use a field button.

We're going to use our provider.isLoading to check if it's loading. Then it's going to be don't

allow it. And it's going to be don't allow it to press the button. Otherwise, we're going to use our

handle send that we already have. And then I'm going to replace the child here with a cons icon.

I can store send. And then I'm also going to add styling to this. So style.

And then I'm going to say it's shape first to a circular border. And then finally said it's padding

to edge insets or edge insets.

And set it to 40.

Now we're also done with our input bar. Let's remove this key as we don't need it.

Let's review what errors we have.

Here I'm going to replace this create full screen with our chat screen that we created.

Let's import it.

And that's it. Now let's run our program.

So I also went ahead and added our provider in the main class.

And I think there was an error here. Let's run this.

Yes, it's not picking up our chat provider. Let's restart it quickly.

And go into our main class. I'm going to attach a debugger here so it knows that this line exists.

Let's wait for it to restart.

Now let's go into our chat screen.

And as you can see now it's recognizing our chat provider and our chat screen is working flawlessly.

So we've successfully built our chatbot UI. Next we are going to start integrating our chatbot

with the Gemini API using Firebase. Thank you for watching and I'll see you in the next one.

12_building-an-ai-chatbot

1. Chatbot UI Part-I

Welcome to this module on building an AI chatbot in Flutter.

AI-powered chatbots are becoming a common feature in modern applications because they allow users to interact with apps using natural language and receiving intelligent responses in real-time.

In this module, we'll explore what AI chatbots are, why they're valuable for mobile applications, and how we can integrate one into our Flutter app.

So what is an AI chatbot?

An AI chatbot is a conversational interface that allows users to communicate with an application using natural language, just as they would with another person.

Modern chatbots are powered by advanced AI models such as Gemini and ChatGPT, which can understand context, generate human-like responses, and assist users with a variety of tasks.

So why add a chatbot to your Flutter app?

AI chatbots provide several benefits for both users and app developers.

They can offer instant assistance and support without requiring users to search through documentations or help pages.

Chatbots also increase users' engagement by creating a more interactive experience and encouraging users to spend more time in the app.

So let's get started on building a chatbot.

So let's start off by creating the UI for our chatbot, for that I'm going to head into my features folder and create a subfolder, the chat screens, and then inside it I'm going to create a chat screen prototype.

Make it a stateful widget, the chat screen.

Now let's replace our placeholder.

I'm going to start by setting my app bar first, so let's remove this, and I'm going to set its background color to our color scheme that we already have.

So color scheme dot primary, and let's import that final color scheme is our theme of color scheme.

Let's rename this.

And then I'm going to set its title spacing to zero.

Let's start by setting up our title, show a circle avatar, which is basically going to be our AI, and replace this by auto awesome to represent our AI.

And then I'm going to set its size to 18.

Let's try finding this, auto awesome.

Then below this, I'm going to add a size box, the size box with a width of 10.

And then I'm going to create a column.

I'm going to first set its cross axis alignment to start, and then a text, which is going to be AI assistant, set up by style, and then text style.

I'm going to change this to 18, and then we want to set it to W600.

Color is going to be white by default, so let's just remove that.

Next let's go ahead and create a provider for our chat screen, so you can work on it simultaneously.

So a new file in our providers, I'm going to call it the chat provider.

And let's create chat provider, which extends our change notified.

I'm going to create a is loading variable here, and then also a getter for this.

So bool get is loading, now let's head back into our app bar.

Now we're going to consume that here, so add a consumer, and let's provide its name.

It's going to be the chat provider, import it, let's rename this to our provider, and then depending on our state, we're going to be showing the status.

So if provider.isLoading, show typing or online.

Let's set its font size to say 12 instead, and then according to our state, we're going to check provider is loading.

If so, then just use the primary color scheme, remove this, otherwise just show that you can type, so let's show it green, and let's add a shade of 600.

So our app bar is all set, now let's move on to the body.

I'm going to replace this with a column, and then use an expanded widget, to each child I'm going to provide another consumer, so let's use the chat provider again.

Next we're going to head into our chat provider, and create a new list for our chat messages.

This is basically what we will be showing to the user, so we don't have this yet.

So next I'm going to go into my models folder and create a new chat model, and I'm going to create an enum here called the message sender, which will basically tell us whether it's the user or the AI that's sending the message.

Then I'm going to create my module class of chat message, first thing we'll have is our

dog ID from Firebase, then we'll have our string, text, and our message sender, which
will basically be the sender, and then the timestamp.
Let me quickly create the tool map, since we've already gone through the basics of this
I don't need to go very in-depth for it.
So shared message ...
So our model is set up, let's go back to our chat provider and import it.
Next let's create a getter for our messages list, so a list of chat messages, and let's
make it an unmodifiable list, because the messages are not going to be changed, the
previous ones, so let's just pass it the message.
So now that we have our messages, let's go back to our chat screen and continue building on that.
Now here the first thing we're going to check is if our messages are empty, so if our provider.messages.isEmpty,
I'm going to return an empty stage from here, and let's pass it our color scheme.
So our class of EmptyStage, let's make this a stateless widget, set up the padding first,
and I'm going to assign a column, and then the first thing I'm going to use here is
an icon, which is going to be icons.AutoAwesome, and I'm going to set its size to 64, and the
primary is going to be with values, let's say alpha, and 0.3 should do.
helpful, how can I help you, and then let's set the style, and then I'm going to use another
text here, us, me, anything.
So now the EmptyStage is set up, let's go back to our main screen, and let's review
what errors we have.
Now from here, since we're returning an EmptyStage in the if block, we also need to return something
else, let's set the item count, so it's going to be our message's length, and then if our
provider is loading, let's set it to 1 or 0, I'm going to set its padding, and from
here we're going to be returning a message bubble widget that we're going to create in
a moment, and to that we're going to be passing a parameter message, and the message we have.
Although if the provider is still loading here, is loading, and if our index is equal
to provider message length, I'm going to return a typing indicator, which is also something
we're going to create in a moment, which is basically going to show that AI is currently typing.
So let's go ahead and create a topic indicator next, below our EmptyStage, I'm going to create
a StageFull widget, the typing indicator widget, I'm going to start by setting its build to
padding, and then set the padding, I didn't set start only, and I'm going to set the bottom

to 12, close this, and then since it's going to be sort of an animation, I'm going to extend it with a mixin, the single decay, which is basically used for animations, as you can see, to create the animations, and then I'm also going to have to create an animation controller here, so animation controller, and then set up my init state, which is going to initialize my controller, and I'm going to set it to 900 duration, after 900 milliseconds, it's going to restart the animation.

We also need to make sure to dispose it, so there are no leaves, so override, and then dispose it, in the build, I'm also going to set my color scheme, so I can use it later.

First, it's going to be a circle avatar, which is basically going to denote our AI, instead of this animation builder, I'm going to use a container, so let's remove this, add a container here, and symmetry, I'm going to set the horizontal to 16, and then the vertical to 12.

And then once we're done with our decoration, I'm going to assign it a child, which is basically going to be an animated builder, which is going to build our animation, I'm going to assign it the controller we created, and then to the builder, we don't need either of these, so let's just skip those, and let's assign it a role, to the children, I'm just going to directly assign a list, so generate, and its length is going to be three, just three doors that we need, to show that AI is loading a response, and then we're going to set our animation delays, so final delay is going to be I by three, then T, I'm going to assign it controller.value minus the delay, and we're going to clamp it, 0.0 and 1.0. Finally, I'm going to set the opacity, you're going to see in a moment how we're using the T variable we created, 0.3 plus 0.7, and this is basically raw code you can put off the internet.

Now, let's use it, return a padding widget from here, so padding,

so now our typing indicator is ready. So in this video, we explored the fundamental of AI chat boards, why we need them, basics on how they work, and finally, how they help scale our application. We'll continue building our chat screen in the next part. Thank you for watching and see you in the next one.

2. Chatbot UI Part-II

Hello everyone and welcome back. In the last video, we were working on our chatbot UI and in this video, we're going to continue on that. So let's get started.

As you can see, let's close this. Our typing indicator is working. Let's create our message bubble next. Let's make it a stateless widget because it's a simple bubble. User doesn't

interact with it. I'm going to call it the message bubble. Let's remove this and I'm going to be passing it our message bubble. The chat message we created. Let's import that here. Okay, so next I'm going to go and replace the placeholder again with our padding widget. So first we're going to check if it is the user or the AI that sent the message. So message sender. Next, we're going to check if it has text or not. So message dot text is not empty. Now let's get started on our message bubble. Let's set the cross axis alignment to the end and then the main axis alignment is going to depend on whether if it is the user or not. So user. If it is, then it's going to be on the right side. Otherwise it's going to be on the left side. So now if it is the user, then we're going to show our avatar. So it's going to be circle avatar. And I'm going to set the radius to 16. Next, let's use a flexible widget so we don't have any overflows. And I'm going to assign it a column. And then let's say the columns cross axis alignment depending on if it's the user or not. So if it is the user, it should cross axis alignment or end. Otherwise, this stuff. And let's say the children. First, I'm going to use a container for the bubble. I'm going to set each decoration to box decoration. Colors can depend on whether it's the user or not. So you can see how our enum is coming in handy now. Because we constantly need to compare whether it is the user or the AI in our chat screen. Now, if the text is present, only then show the bubble correctly. So padding. I'm going to set it to 14. Now let's move on to each child. Assign it a text, the text of the message, the message text. And then set its style. I'm going to set the font size to 15. And the height is going to be 1.4. And then below our text, we're going to show our timestamp for when the message was sent. So let's use a size box first. Height is going to be 4. And then the text, I'm going to use the date that we get from our message. So date format. Let's import Intel. And I'm going to set the format to time. And then format our message to a timestamp. Let's set its style. So text style. Font size is going to be 11. And then the color scheme. Font surface with values of alpha zero point. That's a profile. We set our UI for the AI. Let's check if it's the user. In which case, we're just going to show a size box. And then also the circle avatar. I'm going to set the radius to 16. Background color is going to be color scheme. Secondary. Now we're also done with our message. Bubble. And this is all set. Now we want that whenever a new chat opens or whenever a new message is sent, that the chat is scrolled down to the bottom automatically. So for that, we're going to go inside our list view builder.

And then I'm going to see here the widget bindings after the widget is done building.
It's going to be scroll to bottom. Now let's move above our build. And let's create a method here.
Scroll to bottom. Inside this, I'm also going to check to make sure we do not have any errors.
The widget findings. Instance. Post frame callback after the widget is done building.
Now I'm going to create a scroll controller. Scroll controller, which is aware of the state
of the screen. So scroll controller. And then here, I'm going to check if the scroll controller
has clients. It basically tells the scroll controller if the screen was scrolled automatically
or by the user. So scroll controller.Animate to. And then I'm going to send the offset to
.position.maxextent. Go to the bottom of the screen. Duration, I'm going to set to
milliseconds.300. And then finally, the current is going to be.
Now we have our scroll controller set. Finally, we need an input bar on our screen so the user
can type a message. For that, I'm going to go below my ListViewBuilder, outside the expanded,
so it will be at the bottom. I'm going to create an input bar. To that I'm going to pass the
controller. Scroll controller. I'm going to set a function onSendPrintingImage. You're going to
see how this printing image comes in hand later when we insert images also along with text. For
now, let's just do sendMessage. And then context.read. I'm going to create the sendMessage.
It's going to be needing a chat provider, right? As we said, so provider. And the final text
is going to be controller.text.trim. Let's create design the controller. The controller is going to
be a text editing controller because it's a text box. If our text is empty, don't do anything.
Just return it from here. Otherwise, controller.clear. And then here we're going to do our
createMessage function in our provider. We haven't created that yet. Let's save this.
And afterward, it's going to scroll to bottom.
Now our input bar is pending. Let's go ahead and create that next. I'm going to quickly work
through input bar. Let's make it a stateful widget.
And we're going to need two parameters here that are required. So this.controller and required.
Let's create a text editing controller here.
And then I'm going to create the function.
For now, let's make it an X file for later when we integrate our image.
And then the onSend function. Let's import this.
I'm going to create a handleSend function.
Widget.onSend. Since you don't want to call it directly. For now, let's set it to null
because we won't be sending an image. Let's start on our input bar. I'm going to need a color scheme

I'm going to need a color scheme here. So let's create a variable called that.

Replace this with a safe area widget so it's not hidden

behind the navigation buttons at the bottom of most Android phones. Assign it a container.

And I'm going to set the padding.

Create a child in our container quickly.

Assign it a column.

And set the main axis size to main axis size to a minimum.

And then the children.

So for the input bar, I'm going to use a row and going to use an expanded inlet.

Assign the child a text field.

To the controller, we've already initialized a controller. So widget.controller.

And then let me quickly set its properties.

OnSubmitIt.

The function we've already created.

And then let me just quickly set the decorations.

Okay, now we're done with our input bar.

And then I'm going to create a sample.

Okay, now we're done with our input bar.

And then I'm going to create a send button. So for that, we're going to need a consumer.

Or the chat provider.

So I'm going to use a field button.

We're going to use our provider.isLoading to check if it's loading.

Then it's going to be don't allow it.

And it's going to be don't allow it to press the button.

Otherwise, we're going to use our handle send that we already have.

And then I'm going to replace the child here with a cons icon.

I can start send.

And then I'm also going to add styling to this.

So style.

There's button.

.styleForms.

And then I'm going to say it's shape first to a.

And then I'm going to say it's shape first to a.

And then I'm going to say it's shape first to a circular border.

And then finally said it's padding to.

EdgeInsets.

Or insert it to 40.

Now we're also done with our input bar.

Let's remove this key as we don't need it.

Let's review what errors we have.

Here I'm going to replace this create full screen with a check screen that we created.

Let's import it.

And that's it.

Now let's run our program.

So I also went ahead and added our provider in the main class.

And I'm going to add a new class.

And I'm going to add a new class.

And I'm going to add a new class.

And I think there was an error here. Let's run this.

Ah yes, it's not picking up our chat provider.

Let's restart it quickly.

And go into our main class.

I'm going to attach a debugger here so it knows that this line exists.

Let's wait for it to restart.

Now let's go on to our chat screen.

And as you can see now it's recognizing our chat provider.

And our chat screen is working flawlessly.

So we've successfully built our chatbot UI.

Next, we are going to start integrating our chatbot with the Gemini API using Firebase.

Thank you for watching and I'll see you in the next one.

3. Gemini AI Integration

Hello everyone and welcome back. In this video, I'm going to explain the workflow for connecting Gemini through Firebase to your application.

We're going to connect our Flutter application to Gemini AI using Firebase. This will allow our chatbot to send user messages to the AI model and receive intelligent responses. We'll start by configuring the required settings in Firebase. Then we'll set up our Flutter project so it can communicate with Gemini securely.

Before we use AI features in our applications, we need to enable and configure Gemini in Firebase. We'll go through the necessary steps, make sure our project is connected correctly and install the packages required for AI functionality. Our chatbot becomes much more useful when it understands the application it's part of. That's why we'll provide the AI model with instructions describing how our app works and what features it offers. This context helps the chatbot answer app-related questions more accurately.

So let's go ahead and set up our Firebase API first. For that, I'm going to go into AI services in my Firebase project. I created this sample project because in the initial one I've already set up our AI. So go into AI logic. And then you're going to go ahead and get started. For testing purposes, we're going to go with Gemini developer AI since it is free. And then enable API. And I'm going to disable AI monitoring since we don't really need it for a small project. But it is recommended for larger ones.

Now since that is set up, let's go into our project. And here the first thing you're going to import are these two packages, Firebase AppCheck and Firebase AI to support our chatbot. Now below our Firebase messaging, I'm going to set up my Firebase AppCheck. Firebase AppCheck.instance.activate. And in this, I'm going to set my Android provider. Android provider.debug. And then for iOS, I'm also going to set it up, but it's completely optional, whichever one you want to use. Apple provider.debug. Also make sure your initialize app is set up. I'm going to import this. Save the file. Now you're going to clean and rebuild your projects. So our changes are initialized. So Flutter clean. And then Flutter pub gets. Let's go ahead and run our projects. Connect to this current device and let's start debugging. Go into our debug console. Now here, I'm going to search for debug. And we're going to copy this key into our Firebase. So for that, go into security and AppCheck. And then into our apps tab. Here, I'm going to go manage debug tokens and add a new token. Let's call it token. And after adding it, you can save it. So we're done here. Now our Firebase setup is done. Let's go into our Flutter app. And first, if you go into your Google Cloud console, select your project. And go into your API services and credentials. You'll see that your Gemini developer API key has been automatically generated by Firebase. So that means your setup is working. Now let's head

back into our project. In the provider, we're going to initialize our chat provider first.
The first thing I'm going to initialize here are a few services,
chat firestore service. Then I'm going to initialize late final generative model.
And finally, a chat session. And then initialize our chat provider.
Next, I'm going to initialize my model. Firebase AI dot Google AI and its generative model. I'm going to use the Gemini flash latest. Make sure there are no typos here. Otherwise,
your code is not going to work. Now we also pass it some system instructions. Content dot.
So I already have the instructions. I'm going to paste these here. These basically tell the
AI that you are an assistant for next poll. What you're going to do is create location-based polls.
And if the user asks you to create a poll, just create it with a title and options.
Three options. And keep the title short. Do not suggest images.
So here I'm going to add a chat variable. And I'm going to start a new chat here.
This basically starts a new session. Now let's go ahead and create a chat
firestore service class. I'm going to go into my chat screens and create a new file.
chat firestore service dot. And make a class to save it. And import that here.
We're going to be using this services class for maintaining our chat history.
So that's it. That's all the basic setup you need to connect with your Gemini model or any other model. So in this video, we connected our Flutter application to Gemini AI through Firebase and completed the required setup and configuration. We also provided the AI model with application specific instructions so it can understand our app and deliver more relevant responses to our users.
That's it for this video. Thank you for watching and I'll see you in the next one.

4. Messages Integration

Hello everyone and welcome back. In this video, we'll be learning how to send user messages to Gemini AI and save conversations. Basically, we're going to be implementing the port functionality of our chatbot by creating a function that sends user messages to AI and handles the responses returned from the model. After receiving the AI responses, we'll store both the user messages and the AI's reply in Firebase. This will allow us to keep a record of conversations and use this data later to display chat history when the user returns to the chatbot. Let's get started. So since our basic chatbot is set up, let's create a send message function and a string of text which is going to be the user message. First thing I'm going to check here is if my current child ID is equal to null. Now this current child is going to

be a string denoting our Firebase chat document. If it is null, I'm going to get this text and set it as the first message. Because we're assuming now that no other history has been created. And if it's empty, just send an image. Next, let's create a try-catch block. I'm going to set current child ID is FirestoreService, the class we created, and a createConversation function which is going to create the conversation. So let's go ahead and create this now in our chat FirestoreService. We're going to set up a few variables here first. So Firestore and Firebase Firestore.

And then I'm going to get the odds for getting our users.

Now I'm going to create a helper function to get my user's ID, so UID.

UID is going to be currentOdd.currentUser and it's UID. If this UID is null, and then from here I'm just going to return the UID. Next, I'm going to get a collection reference because our chat history is going to have subcollections to store our messages. String and dynamic. And I'm going to get the chats column. Firestore.collection of users. .doc against the UID..collection of chats. And we're also going to copy this.

Here we're going to create a helper function for getting the messages column. It's going to have this string of chat ID and then using this chat columns that we created, replace it here. And instead of collections, I'm going to access the doc.

Now we can access both our columns and our chat. Now let's go ahead and create our conversation. And I'm going to put it inside this try catch block.

First thing I'm going to extract is the title from our text. So, first message. .length. If it's greater than 40 characters, then I'm going to add a title. And I'm going to add a title.

Then I'm going to get the first message..substring. And just get the 0 to 40 characters. We don't need too many. Next, if it's less than 40, then just copy the first message. The first message. Next, I'm going to get the reference. So, await chats column..add. First the title and assign it. Next, I'm going to get the last message.

So, first message is also going to be the last message..update it at. It was last updated just now. So, timestamp of now down to the millisecond epoch. Finally, send a message count also, which is going to be 0 at this time. From here, I'm going to return retro.

I'm going to return the reference ID that we just created.

Let's load a message here.

Remove this. So, our create conversation is done. Let's head back into our chat provider.

From here, I'm going to pass it the message now. First message is going to be our first message that we created. Next, let's load the error. And if there is an error in the messages list, add a manual chat message. So, we can show the user that an error has occurred or something like could not start conversation.

Sender is going to be message sender.ai. For me, I'm going to call the notify listeners.

So, our chat list is updated and the messages show on the screen. Now that we've set up our reference, let's get ahead of this if. Next, I'm going to get the chat ID from our current child ID. It is not going to be null at this point. So, I'm going to get the user message by creating a chat message. Sender is obviously going to be the user from here. So, message sender .user. Add this to our messages list.

User message.

It's loading. I'm going to set to true now since we're actually starting our process of sending the message to AI. And let's set it notify listeners. Now, this will immediately update the list of messages and the user's message will show on the screen to the user. Next, I'm going to be saving this message into Firebase.

So, let's go back into our Firestore service and create a save message to which we're going to pass our chat ID and then the chat message.

Get the reference and message column and edit message stored to map.

And then I'm just going to return the reference ID. Log the message.

And from here, I'm going to be returning a string which is going to be the document reference.

Save it. Now, let's save the message. Firestore service stored save message.

Chat ID and I'm going to pass the user message.

And if something happens because we're retrying the error.

Now, our message is getting saved to Firebase.

Let's go ahead and send it to our AI.

I'm going to set it in multi parts because later we're going to be integrating images to it also.

So, let's make it portable there from the beginning. I'm going to do part storage ads.

And first thing we're sending it is the text box. So, let's check if it's not empty.

Send it. Otherwise, when we later send it to an image, we're going to attach a query. What is this image?

From the chat, send message. Send message.

Content.multipart and send it the parts.

Get the reply.

And this is mainly the method you'll be using to communicate with your AI.

This is basically going to send a message as a continuation of the chat.

Now that we have our response from AI, let's get the AI message.

Chat message. First thing I'm going to get is the text. So, reply is going to be the text.

Sender is obviously message. Sender.

Timestamp is datetime.now. Now, I'm going to add it to my messages list so we can show it to the user.

Next, we're going to add the AI's response to our Firebase history also. So, let's copy this.

From here, I'm going to set the AI message. I'm going to access the Firestore service and update conversation meta. Let's go ahead and create it.

Update conversation meta.

You're going to see later how it comes handy.

Get the chats column and inside the chat talk,

I'm going to update the conversation.

The last message is going to be the message we just sent.

And then update it just now.

Finally, the message count.

Log the error here, save it, head back into our update.

I'm going to pass it the chat ID, last message, otherwise it's an image.

Finally, catch the error here.

Now our send message is almost finished, let's set up the catch block.

If there was an error, let's add an error message in the list.

Message sender is AI because we had a problem generating the AI message, and date time is now.

Finally, we're going to set his login to false, and finally notify our listeners. Save this.

I'm going to go into my chat screen and hook up the send message function here.

Make it to the provider.sendMessage, type then, and from here scroll to bottom.

Once the message is sent, the new message appears automatically.

Let's go into our chat screen, and let me type up a quick message.

As you can see, our animations are working correctly, and it's showing typing.

And this is a response we received.

It's a generic message that we sent, so it's telling us based on the context we sent it about our app.

Our chatbot is working correctly for basic texts.

Next, we're going to try integrating images into it.

In this video, we created the functionality to send messages from our Fletcher app to Gemini AI and receive responses from the AI models.

We also stored both user and AI messages in Firebase, which gave us a persistent conversation history that we can use later to display previous chats and improve the overall chatbot experience.

So that's it for this video.

Thank you for watching, and I'll see you in the next one.

5. Chat Image Support Part-I

Hello everyone and welcome back.

In this video, we're going to integrate images along with our text messages to AIs.

We're going to expand our chatbot by adding image port.

Users will be able to select images from their gallery or capture new photos using the device camera.

We'll send these images to Gemini AIs so the model can analyze and respond to the visual content.

To ensure image messages are preserved in conversation history, we'll also upload them to Firebase Storage and save the relevant information alongside our chat messages.

So let's get started.

So let's get started on integrating our images.

I'm going to create a new file first, chatStorageService.dart, then I'm going to go into my chatFirestore service and I'm going to copy this because we're going to reuse this. Add this here.

Another thing we're going to need is Firebase Instance.

Now I'm going to start working on my function.

This function is going to upload any image that user sends to our Firestore bucket.

So we can use it to display later in our chats.

Created the iCatch block.

Next I'm going to get the extension of the file.

So final extension and p.extension, import this.

I'm going to pass it the file's name and check if it's not empty.

If so, then get the extension, otherwise if it is empty, set it as .jpg.

Then I'm going to get the file name for storing it.

I'm going to set it to datetime.now and then I'm going to concatenate the extension here.

Then I'm going to get the storage path, set it to something you find easy, but I'm going to set it to the user's ID, chatImages, chatID and finally the file name.

Then I'm going to get the reference to the storage bucket, Firebase Firestore, and then I'm going to pass it my storage path.

Next create the metadata.

I'm going to set its content type first and I'm going to do it mine from extension function that we have to create ourselves.

Let me just add it here.

This is a pretty basic function, it's going to check which file type it is and then just concatenate the image path next to it.

If it's a .gif file, it's going to go into the .gif folder, if it's a .jpg file, it's going to go into the .jpg folder, and if it's just a simple .jpg, it's going to go directly inside.

Next to the content type, I'm going to pass it a custom metadata uploaded by and also the chatID for which chatID belongs to.

Let's move this out here.

Finally I'm going to put the file in the bucket, so put file, pass it the file path and also the metadata.

Then I'm going to return the reference and its download URL.

Make this a string and in the catch block, let me quickly set up an error message and finally I'm going to rethrow it from here.

Now we're done with our upload chat image, let's move back into our chat provider.

I'm going to integrate it inside my sendMessage.

First thing it's going to support is a .x file for image, which is optional.

Then below my chatID, we're sending our simple text message.

We're going to add our image into it now.

Let's upload our image first to Firebase.

Set up a variable for uploaded image URL.

I'm going to check if the image is not null because the image is optional.

Set is loading to true.

Notify listeners.

Set up a try-catch block.

I'm going to upload the image to my storage service.

Final chat storage service, chat storage service.

I'm going to pass it my image and the chatID. Let's await it.

In the catch block, I'm going to set is loading to false and add the message.

In the messages list, chat message, image upload fails.

Let's try again.

Finally notify the listeners.

And I'm just going to return from here.

Since it's failing, we don't need to continue sending it to our AI.

Add the image URL here.

Now the rest of it is set up.

When we're setting our multifiers, I'm going to add my image here.

I'm going to check if my image is not equal to null.

If it is not null, then get the bytes of the image because we're going to be sending

it as bytes and not an entire file.

Image.bytes.read as bytes.

Then I'm going to get the file extension.

Use the same method here.

And then I'm just going to do file parts.add inline data part and bytes.

Let's add the same method here.

And here we've already added.

If user doesn't attach any text, just an image just to test what is this image so AI can answer. And this is it.

Our send message function is done.

Our images are integrated here.

Let's save this and move on to our third screen.

First thing I'm going to add here is image port in our send message. Pending image.

And inside of send message, I'm going to attach the image.

Let's add this in our send message. Pending image.

Next, let's move on to the UI.

Inside my message bubble, I'm going to also include image now.

Check if it has image.

Here I'm going to check if the image is not not. It's not empty.

Then I'm going to go inside my flexible widget.

And before has text, I'm going to check if it has image.

And then I'm going to use a widget we're going to create the chat image with passing

in the image URL.

And then whether it's the user or the AI that's showing this image.

And when setting the padding here, depending on if there is an image or not, let's change it a little.

If it is an image, then it's going to be 8, otherwise going to be 10.

Now let's create our chat image widget.

They just take this widget.

I'm going to call it chat image.

And instead of a sense it's stateless, I'm going to be passing it the image URL. And is user.

Let's create these next image URL and is user.

I'm going to replace this with a constraint box, set its box constraints, max height and

then max width is going to be depending on the size of the screen.

I'm going to get the size and its width.

And to its child, it's going to be an image network, box fit, don't take all the remaining

space or the available space.

And then while it's loading, I'm going to get the child and the progress.

If progress is now, then return child because it has been loaded.

Then I'm going to return a center region from here, which is going to show a progress

indicator that the image is loading.

And to its value, I'm going to set the progress expected are not now.

So the progress, those cumulative bytes that have been loaded divided by how many are left

to expected total bytes, otherwise just now.

Set the stroke width to two.

Then I'm going to build the error builder.

If there is an issue, we're just going to show this.

Height is going to be 80.

And then the child is going to be a center region, which shows an icon.

I can start broken and set the size to, let's see what errors we have.

Let's make it non-nullable.

And this is it for part one.

In the next part, we're going to do the rest of the integration and then see if it's working or not.

In this video, we enhanced our chatbot with image capabilities by allowing users to send

photos from their gallery or camera.

We also integrated Firebase stories to store uploaded images, ensuring they can be displayed

later as part of the conversation history.

This gives our chatbot the ability to work with both text and visual experience.

In the next video, we'll continue building on our image integration.

6. Chat Image Support Part-II

Hello everyone and welcome back.

In the last video, we started our image integration in our chat with AI.

So let's continue to build on that and get it working.

So now that our chat image is done, let's move ahead into our input bar.

And we're going to start integrating image here, we've already created the function.

Next I'm going to create an X file here, pending image that has not been uploaded yet.

And then create a picker.

Next I'm going to create a pick image function, which is going to be a future of voyage.

Pick image from our image source, the source, and then make it async.

Now I'm going to check the picked file is picker.pickimage.

To the source, I'm going to provide it the source variable we created.

Image quality is going to be, say, 85, because we don't want the images to be too heavy when

we're sending it in bytes, and then 1920 maximum weight.

Next I'm going to check if the picked image is not null, update the widget, update this.

Next I'm going to create a function for showing the model sheet, our model source sheet.

In the safe area, I'm going to assign a padding, padding search symmetry, the vertical is going to be 12.

Inside the padding widget, I'm going to set a column, main exercise is going to be .min,

and then children.

I'm going to set the shape of my model bottom sheet first, so a rounded, rectangular border.

And then I'm going to set the border radius to a bit more than is by default, border radius .vertical.

From the top, I'm going to assign it a radius .circular to 20.

Let's move back into our column.

First I'm going to use a container as the top handle of my model sheet, so user can

drag, notice that they can drag it to close it.

Margin only, and I'm going to set the border.

Similarly set up the decorations, color scheme, then the border radius, then below my container

I'm going to create a list style, leading is going to be icon, photo, library, outline.

Then I'm going to set its title to a text, choose from gallery, and on top of this style, navigator.pop to close our bottom sheet.

Then I'm going to call the pick image function, and provide it the source, which is going to be image source.gallery, as selected by the user.

Let's copy this.

I'm going to use a camera, and instead take a photo.

And from here I'm going to provide the source as the camera.

Now outside this function, I'm going to create a new one, set the pending image to null.

Next inside my handle send, we're going to be including our image now instead of a null, the final image is going to be the pending image.

And then I'm going to clean my state by setting our pending image to null.

Finally I'm going to send the image from here.

So it's removed from our, now I'm going to integrate the image into our input bar.

So above my text row, I'm going to check if the pending image is not null, in which case I'm going to show a padding widget, edge and switch only, and the bottom is going to be it.

Set the child to align, and alignment.

Next I'm going to get a stack for showing the image above the input bar.

So clip behavior, clip to none.

And then I'm going to set the children, show the thumbnail here.

I'm going to use clip Rx for that, and set the border radius.

And to the child, I'm going to assign the image file that we have, pick for the user.

To set the file, dot path, and I'm going to set its width and height.

And then it is going to be, now on the image, I'm going to add a button to remove it.

So position from the top, set it to minus eight, and from the right, also.

Now I'm going to get the child.

I'm going to assign it a gesture detector.

On tab, we're going to use our clear image function that we created.

Next I'm going to assign the child, it's going to be a container with an icon in it.

I'm going to set the width and height, and then the decorations, box decoration, reset the color, and then the shape is going to be box shape church circle, which has the border.

We're going to make the border a bit darker, the surface, and width is going to be two.

Next to the child of the container, I'm going to assign the icon.

I can start close, size is going to be 12, and color is going to be color scheme, dot on.

So our preview of the image is done.

Let's move into our row, and here, before the expanded widget, I'm going to add another icon button.

And here, we're going to use the show image, source sheet, so when the user clicks on it,

the source sheet appears.

Icon is going to be, I can start attach, file, and its color is going to depend on whether

the image is attached or not, so if the pending image is not equal to null, color scheme dot

primary, otherwise, color scheme, dot on surface, dot with values, and I'm going to

set the alpha to 0.5.

Now our image attachment is also done.

Let's set up our index.

If there is a pending image, add a caption, optional, otherwise, just show the text, okay,

so now our image integration is done.

Let's see if send message is set up correctly, and yes, we're passing in the image, so let's

go ahead and run our program.

Let's head into our chat, and as you can see, our UI is working.

We have our attach image option, let's attach an image, and send it.

It seems it's not being sent.

Let's check our send function.

Here we need to also add that if both text and image are null, only then don't send it.

Let's save this, and try uploading an image again.

I'm going to send it without the text.

As you can see, add a caption, the index has also changed.

Now our button has been disabled, and AI is typing.

We can also see our image preview.

As you can see, it's providing a description.

Let's ask it to create a poll based on the image, and send it.

So it has successfully created a poll after doing our image analysis.

So in this video, we successfully completed our image integration.

Now our chatbot fully supports images along with text, and provide responses after image analysis.

If you want, you can also set up image generation by AI for practice.

That's it for this video, thank you for watching, and I'll see you in the next one.

7. AI Chat History

Hello everyone and welcome back. In this video, we'll work on our chat history. We'll start by building a chat history screen for our chatbot. We'll use Firebase Streams to listen for changes in real time and display a list of previous conversations. This ensures that any new or updated chats appear automatically without requiring a manual refresh. We'll also add the ability to delete conversations, giving users an easy way to manage and organize their chat history. Let's get started. So I've already set up this basic history screen. I'm going to set up the build in a second, but I've created this open new chat and confirm delete dialogues for creating and deleting chats. And then I've also created this conversation tile UI, which is going to display each chat as a tile on our screen. It's basically going to display the time on each tile and then the conversation, the last message we had. And then finally, at the end of the tile, we're going to have how many messages are in that conversation. And after that, I've also created this empty conversations widget. It's going to show a text that no conversation have been started yet. So user can start a new one when there are no previous chats available. So let's go to our build section and I'm going to go to my open new chat and I'm going to need to create this method. Let's import chat provider first and then also provider. Now I'm going to create this reset for new chat. It's basically going to be used for if a previous chat was already loaded. This is going to clear everything up before that. Let's go ahead and create it. Below my init, I'm going to create a reset for new chat. First, I'm going to reset the current chat ID, set it to null. Then I'm going to set my messages list to empty. So clear it. I'm going to start a new chat through the model. So start chat. So it doesn't have the context from our previous chats. And finally, I'm just going to notify the listeners that it's empty. So that's it. Now our new chat is set up. Let's head back into our builds. And here I'm going to start replacing the container. In our build section, we're basically using an app bar to show AI chats. And then we're also showing a floating icon. So user has the option to create a new chat. I'm going to go ahead and hook up my open new chat to my floating button and pass it the context. Now I'm going to start by replacing the container. We're going to use a stream builder here. So we have the most recent data available. Let's make this a list of chat conversation. I've also created this chat conversation model. It's basically going to have our ID title, last message and message count for just displaying it on our chat history. Instead of using the same one, which is going to have multiple chats in it in the chat model,

I just created one for displaying against the tile. And for the stream, I'm going to need the Firebase chat stream. So I'm going to create a new variable service here. Chat Firestore service. Now through the service, I'm going to access a new method dot conversations stream. Now let's go ahead and create this in our chat Firestore service. Below my create conversations, I'm going to create a new stream, which is obviously going to return a list of chat conversations. Import that conversation stream. I'm going to return from here, a chat columns, don't order by get the latest ones. So first, I'm going to sort it by updated at and then send this ending to true. And then I'm going to get the snapshot and I'm going to map them, get each snap. And then snap.docs. First map each one. Get the chat conversation model to map. I'm going to pass it with the data and its ID. Set it to to list. Now it's going to get all the chats, and then sort them by updated at this ending true. So the latest one become on top. And then I'm going to get the snapshots and each conversation. So save this. Now inside the builder, I'm going to pass it the context and the snapshot. And if the snapshot connection state is connection state dot waiting, which means it's still waiting for the result, I'm going to return a center widget with a child to show a circular progress indicator. Now after it's done waiting, it's going to get the snapshot. Let's check if it has the error. In which case I'm going to return another center widget. And through its child, I'm going to set a text could not load, could not load chats. And then get the snapshot.error. Now if there is no error and it's done loading, I'm going to get the conversation snapshot.data. Get the conversation is empty. Let's just return our empty conversations we did that we've already created. And I'm going to pass it the color scheme. And then to the new chat, I'm going to pass it to private. Open new chat function. Now from here, if there is data, let's return a list view. Let's return a list view dot separated to the item builder. Let's pass it the context and index. We're going to need this. I'm going to get the conversation at this index. So conversation to the separator builder.

Let's set these to nothing and just pass a divider.

Let's set its height to one and index to 72.

Item count is going to be conversations length. So dot length.

Set the padding here first. So get the edge symmetric. I'm going to set the vertical to eight.

Now from here, I'm going to return my conversation tile.

That's going to show the details of our conversation. Passing the columns that we created.

And for now, I'm going to set these to now. And later, we're going to set these functions up

to this one since we've already created a delete

confirmation. Let's pass it that. And close this. Now our display is set up. I'm going to go into

my delete function. In the daily dialogue, I'm going to set up a navigator

dot box. And then a try catch block.

Now here, I'm going to create my service. So it's delete conversation.

And we'll be passing in the conversation ID. So let's go ahead and create this in a service class.

Below my conversation stream, I'm going to create a new function.

Delete conversation. And let's pass it the chat ID.

Let's set up a try catch block. I'm going to get the snapshot from await messages

column. Pass you the snap and get. Let's make this async.

And now I'm going to go ahead and delete images from docs.

So when we're creating a chat, we're also sending images as we previously integrated.

So once we're deleting that chat, we also need to delete the images associated with it.

So we also need to create a method for that. Go ahead and create this.

Go ahead and create this.

While delete images from docs, we'll be passing it a list of

query documents snapshot, which is going to be a map of string and dynamic.

I'm going to call it the docs.

Create a new variable deletions, which is going to show the progress. It's going to be a future void.

Now we're going to have to recurse to our documents. So docs.

This is missing.

Now I'm going to get the URL from the doc data, the image URL as a string.

Now I'm going to check if the URL is not null and it's not empty.

Let's use the deletions function. And add a storage.

Now I'm going to need to go into my chat storage

service and create a new method here to delete images from chat.

So let's do that. Delete image function.

And I'm also going to need to create a new method here to delete images from the chat.

So let's do that. Delete image function.

Give it the string download URL.

Set up a try catch block. Make it async.

And through the storage here, I'm going to get reference from URL. Pass it the download URL.

And delete it directly.

And add a log inside my catch block.

Copy this. Also, instead of just catch, I'm also going to check for five ways exception here.

And we're going to catch it if the code is not equal to object not found.

I'm going to log an error.

Copy this. And set up the error.

If there is an object not found error, it's simply going to not do anything because the

image doesn't exist in the first place. Let's go back into our chat firestore.

I'm going to create this storage service variable.

Final. And then I'm going to get the chat storage.

Now let's go back into our delete images function.

And finally, I'm going to check if deletions is not empty.

Then that means we need to wait for it to finish. For the deletions.

Let's head back into our delete conversations function.

Now that our image has been deleted, let's go ahead and delete our conversation also.

I'm going to batch delete it. Firebase.

Firestore.batch. I'm going to get the top in. And then I'm going to delete the entire reference.

It's also going to delete any sub collections.

Outside of my loop, I'm going to go batch delete.

Chat column. Dot doc. And the chat ID.

Await. And then I'm going to just commit these deletions to Firebase.

Let's log an error message.

Chat. Firestore. Conversation. And finally, read through it from here. We're going to catch it.

Let's save this. And head back into our conversations link.

Our conversation screen. In the cat section, I'm going to check if my context is still mounted.

The screen has not been cleared. Show a scaffold. Messenger. Pass it a snack bar widget.

Content is going to be text. Now our chat history screen is finished. Let's quickly go through.

Now it's set up. Let's save this. And run our file.

So, let's go into our plus. I've replaced this with the chat history screen. As you can see,

all our chats are displaying now. We cannot yet go to the details of the chat. We'll set

that up in the next image. Let's try deleting a chat. Delete it. And as you can see, it was

deleted from the chat history screen. So, let's go back to our chat history screen.

Let's try deleting a chat. Delete it. And as you can see, it was deleted successfully.

So, in this video, we created a dedicated chat history screen that displays conversations in real time using Firebase Streams. We also implemented the ability to delete previous chats, making it easy for users to manage their conversation history and keep the interface organized. That's it for this video. Thank you for watching, and I'll see you in the next one.

8. Loading Chat Window with Previous Messages

Hello everyone and welcome back.

In this video, we'll be learning how to load and display our previous conversations with AI.

We'll implement the functionality to load previous conversations from Firebase.

When a user opens an existing chat, we'll retrieve all the saved messages and display them in the chat interface.

This allows users to continue conversations from where they left off, creating a seamless and consistent chatbot experience.

So the first thing I'm going to change here is in my show poll screen, I'm going to go into my floating action button, and I'm going to replace this with the dial, the speed dial.

For this, all you have to do is install the package.

Let's import it.

I'm going to add the icon, icons.add, and through this, we can show all our screens regarding AI, and I'm going to set it icons.close.

If the icon is active, showing all of our screens, colors.orange.

And the foreground color is going to be colors.white.

Active background color is orange.

And the shade, 700.

And then active foreground color is colors.white.

Next I'm going to set the elevation to six, and the spacing between, and the spacing between the button is going to be 12.

And space between the children is going to be eight.

Overlay opacity, I'm going to add a little bit, so 0.4 is enough.

And finally, tooltip, if the user presses on it, I'm going to show actions.

Next I'm going to set the children, a speed dial child, and to the child, I'm going to assign an icon, icons.add, chart rounded, set its background color, colors.orange.

And then the foreground, colors.white.

The first one is going to be for our create poll screen, so let's name it that.

Let's set its label style to text style.

And then font width is going to be font width, .600.

And then font size is 14.

Next I'm going to set the on tap, and I'm going to push the navigator, .push, set the

page transition, page transition.

To the child, I'm going to pass the create poll screen.

Let's import it, and pass it the current ID, current user ID.

And then finally its type, I'm going to set to page transition type.page.

Next I'm going to copy this for my next button.

It's going to be our AI chat, so I'm going to use the chat bubble icon, round it.

Let's make it deep purple.

And instead of create poll, new AI chat.

And then inside my on tap, first I'm going to context.read for my chat provider.reset for new chat.

So if previously any chat was open, its data is erased.

Let's import this, and the read, also here.

And then I'm going to replace this with my chat screen.

Remove this, chat screen, import it.

Next let's add our chat history screen also.

Replace this with history, round it.

This is going to be teal, chat history.

And replace the screen with conversations.

So now our home screen is set up.

Let's go into conversation screen.

Here I'm going to create a new method, write, open, existing, chat.

Build context is going to be context.

And also we'll be passing in the chat conversation. So con.

I'm going to copy this, add it here.

And then also our navigation.

Now I'm going to be changing a few things here.

The first is instead of this reset, I'm going to use load conversation, which we're going to create now.

So let's go into chat provider, and create a new method, load conversation.

I'm going to pass in the chat ID.

First let's check that the current chat ID is equal to chat ID.

If it's the same, there's no reason to reload the conversation.

And then I'm going to set the current child ID to chat ID.

Next I'm going to clear my messages list.

And also I'm going to set a variable, is loading history, to true.

Let's go ahead and create it.

Variable is loading history is equal to false.

Make this private.

And back into our load conversation.

And we're going to notify listeners.

Let's create a try catch block.

I'm going to first check the history, get it from firestore service dot load messages

and pass it the chat ID.

Let's go into our chat firestore service and create this method.

It's going to be returning a list of chat messages, load messages, pass it the chat

ID, set up a try catch block.

And this snap, I'm going to say, await it through messages, column, order by, timestamp,

and finally get it.

And from here, I'm simply going to do snap.docs, map it first, chat message dot from map.

I'm going to pass it the D dot data.

And also the doc ID, which is going to be the ID, finally set it to to list.

And then in the catch block, I'm going to log my message, chat firestore service dot load message.

And from here, I'm going to return an empty list.

Let's save this.

I'm going to add these messages to my messages list dot add all and pass it the pass it the history.

Next I'm going to build up the chat.

And from the history, I'm going to pass it the messages.

So it has the context of our previous messages.

I'm going to get the text.

Text part is going to be get the text part.

M dot text dot is not empty.

If there is text available, otherwise, it's probably an image.

So just send image, we can be sending byte every time because it's going to take up a lot of tokens.

And then after this, I'm going to return it m dot sender message sender dot user.

I'm going to send it the content as a multi-part text part, otherwise send it a content dot

model and send the text parts.

Finally, I'm going to set it to to list.

Set this to colon.

So we're sending a context to our chat bot because there is no point of saving the history

in Firebase if we're not going to let the AI know what the previous conversation was.

This is basically called the context or the history of the chat.

Next I set up the cache block, load the message, chat provided or load conversation.

And finally, after you're done with all of that, set is loading history to false and

notify listeners.

Now save this and return to our conversation screen.

Here I'm going to pass it my conversation dot ID.

Let's head into our list view builder that we've already created.

And instead of this here, I'm going to pass it my open existing chat.

Let's go back inside our function.

And now to our chat screen, we're going to be passing a chat ID, conversation ID, and then a title.

Conversation dot title.

Let's go ahead and add this to our chat screen.

Let's create the variables.

Final string is going to be a chat ID.

And also, both of these are optional because we could be loading a previous chat or creating a new.

So these are not required.

Simply do this dot title and chat ID. Save this.

Go back to our conversation screen.

Move this constant.

And now our conversation history is also set up.

Let's go ahead and run our project.

So let's head into our plus icon.

And we can see all our buttons here, chat history, the new II chat, and finally the
create poll screen.
So let's head into our create poll screen.
And it's navigating successfully.
The new II chat.
And finally, our chat history screen.
So here I'm going to try to go to the new I chat.
And then go into my image.
And as you can see, our image chat is also being loaded successfully.
So in this video, we added support for loading previous conversations from Firebase.
By retrieving and displaying saved messages along with images, user can reopen any chat
and continue the conversation exactly where they left off.
That's it for this video.
Thank you for watching and I'll see you in the next one.

9. Steamline Poll Creation using AI

Hello everyone. In this video, we're going to be streamlining our AI poll generation.
We're going to make our chatbots even smarter by allowing users to generate polls directly
from an image. Instead of requiring the users to type a prompt, we'll automatically send
the image to Gemini AI along with instructions to create a poll that fits our application's
format. We'll then pass the AI's response into structured poll data and present it to
the user for review. If they're happy with the generated content, they can create the
poll with a single tap, making the creation process much faster and more convenient.
So let's start with our sendMessage function. And here you can see if the user uploads an
image but doesn't include the text, we automatically include what is this image. Now below, we're
adding a message. I'm going to create a new poll, which is going to be a parsePoll function.
And we're going to be passing it the reply. Let's create a lastParsePoll first. For that,
we're going to need a parsePoll module, lastParsePoll. Let's import it and go into our model. I've
already created this model. It's going to have the AI-suggested poll data that's parsed
from our chat message. And it's only going to have exactly three options in the list.
So let's head back into our messages. I'm going to create my parsePoll function. It's
going to return the parsePoll, optionally. parsePoll. And to this, we're going to be

parsing the text, set up a try-catch block. I'm going to be using regular expressions to see if there's a poll here, this regular expression, and make sure it's not case-sensitive. It's going to first check the response from AI, and this is how it will be sending us the response. Now I'm going to check if the title doesn't match. If the title is null, which means it did not match, we're going to return null from here. And we also check the first match is the text. Now I'm going to get the title for my title match. I'm going to get the first group. So group one, and I'm going to trim this so there are no whites. Next I'm going to get the options. And here we're going to have three options. So I'm going to set i to one, i is less than three, and then increment it. Next I'm going to get the option match, and get the first match here also from the text. And here I'm going to add my regular expression. Let's check if our option match is null. If so, return this because the poll was not parsed successfully. Otherwise, to my options list, I'm going to add a pitchy match and the group one and trim this. Finally, we're going to return the parsed poll from here. I'm going to set the title and the options. And from here, I'm simply returning null because we were unable to parse the poll successfully. Let's head back into our send messages. Copy the name. Now it's going to automatically parse the poll. Let's create a getter for our last parsed poll. Copy this. And I'm going to set last parsed poll is save this file. Let's go into our chat screen. And from where we're sending our message, I'm going to check here if the provider dot last parsed poll and if this poll is not null, and the context is mounted, I'm going to do provider dot clear last poll. Let's create this function. Copy. And in my chat provider, I'm simply going to set it to null. So, function. And set my last parsed poll to null. So, it's clear for the next poll. Next, I'm going to show a dialogue here. And parse the poll. Now, we're going to show this dialogue to our user to ask them if they automatically want to add the poll that I created and create it automatically. Going to get the parsed poll. Import it. Copy the color scheme. And then I'm going to show a dialogue. It's going to be returning a boolean, whether the user confirms or not. And to the builder, I'm going to assign dialogue context. And then I'm going to set its shape. And then the border radius. I'll parse it 16.

Next to the title. I'm going to parse a row.

And then to the children, let's assign an icon.

Icons dot poll outlines. And then I'm going to set the colors.

Color scheme dot primary. And set the size to 22. Next, I'm going to add a size box.

And set the width to 8. Because it's a row. And finally, a text.

Add this poll. And we're going to show the details of the poll. So, I'm going to set

the content of the dialogue. Set it to a column. Because we're going to be showing multiple

options. I'm going to set main axis size to minimum. Cross axis alignment to start.

Set the children. First thing we'll be showing is the text that the AI created a poll for you.

You want to open the create screen with its pre-filled. Then add the size box again.

And to the height, set it to 12. Next, get the preview card. So, I'm going to use a container.

Set the width to double infinity.

Padding. Add in sets. And I'm going to set it to 12. And finally, quickly set up the decorations.

Set the color to color scheme. And then set the border radius.

And I'm going to set the children to column. And set its cross axis alignment to start also.

Set its children. And the first thing we're going to show is the title of the poll. So,

title. And then its style. Pond weight. Next, I'm going to show a size box.

And set the height to six. Finally, use a loop to show all of the options.

Let me quickly set this up.

All options. And its length, which is always going to be three. So, you can hard code it also.

Now, I'm going to show padding widget. And set it to only

three. For the top. So, each option has some spacing. And then to the child, assign a text.

I plus one. Option number. And then the option. And the style here. Textile.

I'm going to set the font size to 13. And the color. Color scheme. Dot on surface.

And read the values. I'm going to set the alpha to 0.75.

Add the actions. So, you can choose whether to edit or not. I'm going to add a text button first.

And set its on first. Navigator dot pop. And close this dialog.

And set it to false. Also, its child. Also, its child. Just to sign a text. No. Let's copy this.

And instead of a text button, I'm going to use a field button here.

And from here, we'll be returning true. Where the user wants to close it.

Whether the user wants to create the poll. Now, I'm going to check if it is confirmed.

It's true. Which means the user does want to create it automatically.

And then the context is mounted. I'm going to context or treat. My poll provider.

Imported. And then I'm going to use the prepopulate function. And pass it my poll.title.

And then poll.options. Next, I'm going to get the user's ID.

FirebaseAuth.instance.parentUser.uid. If it's empty, just pass it an empty string.

Now, I'm going to do navigator.push. And to the route, I'm going to pass it a material.

Page route. And then get the create poll script.

I'm going to pass this. The parent user ID that I just set. Let's head into our poll providers.

I've already created this function with prepopulates. We'll pass it our title.

And our controller options. From before. And also, our location.

Let's go ahead and run our program.

One more thing we need to change here is go to our chat provider.

And here, instead of what is in this image, the query we're going to send to AI is creates a poll relevant to this image. So, AI automatically creates a poll that is relevant to our image.

Let's go into our AI chat screen. And I'm going to attach an image from our gallery.

The same as press on. And I'm going to send it without a caption.

As you can see, it's asking us if we want to automatically add this poll. Which means AI created it successfully. And tapping on okay takes us to our create poll screen with our data prepopulate. So, in this video, we built an AI powered poll generation feature that creates polls directly from images. We also passed the AI's response into our application's poll format and allowed users to review and create the generated poll with minimal effort.

Providing a faster and more streamlined user experience.

That's it for this video. Thank you for watching and I'll see you in the next one.

13_monetizing-with-google-admob

1. Introduction to Google AdMob

Hello, everyone, and welcome back.

Throughout this course, we focused on building a complete Flutter application with Firebase.

We've implemented authentication, cloud storage, databases, notifications, offline support,

and many other production-ready features.

Now it's time to answer an important question.

How do we start earning revenue from our applications?

One of the most common ways developers monetize their mobile apps is by displaying advertisements.

And Google's AdMob platform is one of the most popular advertising solutions available

for both Android and iOS applications.

In this module, we'll learn how to integrate AdMob with a Flutter application and display

banner advertisements using the official Google Mobile Ads SDK.

The great thing about this module is that you do not need an AdMob account to follow along.

Google provides official test app IDs and test ad unit IDs that allow us to integrate

and display test advertisements safely during development.

This means everyone can complete the module without waiting for account approval or creating

an AdMob account first.

Our primary focus is understanding how the integration works.

Once you understand the process, switching from Google's test advertisements to your

own production advertisements is very straightforward.

Before we start integrating AdMob, let's take a moment to understand the two ad formats

we'll be implementing throughout this module.

The first is the banner ad.

Banner ads are small advertisements that appear within the application's interface, usually

at the top or bottom of the screen.

Since they occupy only a small portion of the display, users can continue interacting

with the app while the ad remains visible.

They are a great option for generating consistent revenue without significantly interrupting

the user experience.

The second is interstitial ads.

Unlike banner ads, interstitial ads take up the entire screen and are designed to be shown at natural transition points within the app, such as after completing an action or when navigating between screens.

Because they capture the user's attention, they generally generate higher revenue, but they should only be used thoughtfully to avoid creating a poor user experience.

In this module, we'll start by configuring our Firebase project for AdMob on both Android and iOS devices.

We'll initialize the Google Mobile Ads SDK, integrate both banner and interstitial advertisements, load and display them within our application, and learn how to properly manage their lifecycle, including loading, showing, disposing, and handling ad events.

By the end of this module, you'll have a fully working implementation of both banner and interstitial advertisements using Google's official test tags.

More importantly, you'll understand exactly what changes are required to replace the test IDs with your own AdMob credentials and begin serving real advertisements to monetize your applications after publication.

And that brings us to the end of this introduction to Google AdMob and what we'll be covering throughout this module.

In the next video, we'll begin by setting up our Firebase project for Google AdMob, installing the required package, and configuring both Android and iOS so that we're ready to start displaying ads in our application.

Thank you for watching and I'll see you in the next lesson.

2. Displaying Banner Ads

Hello everyone and welcome back.

In this lesson, we're going to begin integrating Google AdMob into our Flutter application.

We'll start by adding Google Mobile Ads package to our project and then configure both the Android and iOS platforms using Google's official test FIDs.

These test IDs allow us to safely develop and test our application without creating an AdMob account or serving real advertisements.

After that, we'll load the banner, listen to its loading events, and display it on the home screen.

We'll also ensure the banner is properly disposed of when it's no longer needed to avoid memory leaks.

So let's get started.

So the first thing I'm going to do here is go into my poll provider and I'm going to

create a new function to load a banner ad.

So I'd load ad and it's going to need the following parameters.

First is obviously going to be the build context.

And after that, we'll have screen width.

You'll see how that comes in hand later.

Next I'm going to set up a try catch block.

And the first thing I'm going to get here is the banner ad.

So let's go ahead and create this variable, a banner ad and banner ad.

And for this, you're going to need the following package, Google Mobile Ads.

So make sure to add that first.

Let's go back into our function.

And our first action will be to dispose of this.

So if there is any previous ad, when this is loading, it becomes disposed. Set it to now.

So there's no reference.

Finally, I'm going to get the size, a weight and size dot.

I'm going to get the get large.

To this, I'm going to pass the media query, dot size of.

And then get the width and make it async.

Now this method is only available in the latest version.

If you're going to your Google Developers AdMob documentation, you'll see that Flutter

version 3.38 is required.

So you will also need to set this up if you want to use the latest methods.

Otherwise, you can just use another one, which is get current orientation.

That works on previous version.

Now I'm going to check if the size is now, for some reason, then I'm just going to return it from here.

Finally, I'm going to get my ad and set up a banner ad object.

To the size, I'm going to assign the size we have.

Ad unit ID is going to be banner ads unit ID that we will assign in a second.

To the listener, I'm going to pass a banner ad listener.

And to the request, it will be a constant ad request.

Now for the ad ID, I'm going to pick it up from Google.

I'm going to go to Google Developer, and then test ads.

I'm going to go inside test ads, and you're going to go into Flutter, or either way.

And from here, I'm going to copy this anchored ID, or you can also copy the inline one.

They're both same.

Google provides this so you can use these to test.

Otherwise, if you use your actual IDs after having set up an AdMob account, without setting it to test, Google will block your ID.

So I'm going to set it to banner unit ID and paste that here.

Let's go back inside the function.

And now to the banner ad listener, we're going to check its states on ad loaded.

To load the ad, I'm going to assign it to my banner ad as a banner ad.

And finally, I'm going to notify listeners.

And below my creation variable, I'm going to call ad.load.

Let's go into the catch block.

And here I'm just going to set up a log, poll provider, error, loading, banner, ad.

So here we're basically first disposing our ad.

And after that, we're calculating the size of our ad through the Google mobile ad SDK.

It calculates the ID of banner height based on the current screen width and device.

This creates an anchored adaptive banner, which looks better across different screen size.

Then after that, finally, we're calling ad.load.

This sends the request to AdMob.

Loading happens asynchronously.

And one of the listener callbacks on ad loaded or others, which I'll show you later, will be invoked when it completes.

Now let's show this ad in our show poll screen.

Here I'm going to replace this tab bar.

First, I'm going to wrap it in an expanded widget.

Then I'm going to wrap it inside.

And finally, I'm going to wrap it in a safe area so it doesn't hide behind a navigation button.

Now below my expanded, I'm going to show my banner ad to a consumer of poll provider.

Let's get the context and the provider.

And we don't need this.

I'm going to get the banner ad from my poll provider.bannerad.

And if it is now, just return a sized box.shrink from here.

Next I'm going to go ahead and return an actual sized box with content.

Get the width, banner, ad.size.width, and to double.

Going to do the same for height here also.

And then finally, set the child to an ad widget.

The good thing about this SDK is that it loads the ad asynchronously, so you don't have to show any errors or anything else.

So since our UI is set up now, let's save this and head inside our main class.

And here we're going to initialize our SDK.

So mobile ads.instance.initialize. Save this.

And now I'm going to go inside my Android folder.

App, source, main.

Inside my Android manifest, I'm going to add a new metadata tag. Create it.

Make sure there are no typos here.

Otherwise your project is not going to compile it properly.

Then I'm going to get Android name.

Set it to com.google.android.gms.ads.

And finally, application ID.

You don't need to replace this with your own ID.

This is a static string.

Next I'm going to set its value to this.

You can copy this off the internet also. So save this.

Next I'm going to go inside my iOS runner.

And then info.plist.

Here I'm going to attach a new identifier. Create a key.

And let's close it first.

And then inside this, I'm going to create GAD application identifier.

Then I'm going to create a string. Close it.

And copy and paste the same ID here.

So now our setup is also done.

All you have to do is initialize it in the main.dart file and then set the keys in our

Android and iOS folder.

Now later you're going to replace both these keys with your official AdMob keys for protection.

But for testing, we're going to use these keys that Google has provided us.

Let's close this.

And let's go ahead and run our app.

Let's wait a couple of seconds to see if our ad pops up.

We forgot to call our loadAd function.

So let's go ahead and do that on init.

I'm going to do this inside widget.

Widget bindings.

Add post frame fallback.

And I'm going to call my loadAd here.

Let's just do this through context.read and pollProvider.

Finally, I'm going to pass it the context and then my mediaQuery.of.

And instead of doing it, I'm just going to send it the width.

Let's rerun our app.

And as you can see now, the ad is popping up successfully.

It's a test banner ad.

It says you've loaded a test ad from AdMob.

It's a bit hidden behind a floating action button, but we'll fix that in the next video.

We successfully completed the initial Google AdMob integration into our Flutter app.

In this lesson, we configured both the Android and iOS platforms, initialized the Google

Mobile Ads SDK, and displayed our first banner advertisement using Google's official test ads.

We also learned how to properly load and dispose of a banner ad to ensure efficient resource management.

That's it for this video.

Thank you for watching and I'll see you in the next one.

3. Banner Ad Enhancement

Hello everyone and welcome back.

In the previous lesson, we successfully integrated Google AdMob into our Flutter application

and displayed our first banner advertisement using Google's official test ads.
While our integration is functional, there are still a couple of improvements we can make before moving on to other ad formats.
In this lesson, we'll focus on making our banner implementation more production-ready.
The first improvement is adding analytics.
We log important banner events using Firebase so we can monitor how our ads are performing.
Tracking events such as successful ad loads and failed load dumps can be extremely useful when debugging issues or understanding how ads behave once your application is in the hands of the user.
The other improvement is related to our application's user interface.
Since the banner ad is displayed at the bottom of the screen, our floating action button currently overlaps with it, which isn't a great user experience.
We'll update the layout so the button is positioned above the banner, ensuring that both the advertisement and the application's control remain fully visible and accessible.
So let's get started.
So to set up my banner ad, I'm going to go inside my poll provider.
And we've already set up our onLoaded function.
I'm going to set up a couple more here, such as the onAdFailedToLoad.
And now here, along with disposing our ad, I'm going to create a new method that will load our ad event to Firebase.
Let's go ahead and create this void loadAdEvent.
And then it's going to have a couple of parameters.
The first is going to be a string event, which is required.
And then after, I'm going to have a required string ad type, because even though we've already integrated banner ads for now, we're going to be adding interstitial ads later also.
So string and dynamic.
And an extra map, in case we want to add some other values.
And now let's get started on the body of the function.
I'm going to set up a direct cache block.
And the first thing I'm going to get here is the user ID, firebase.instance.currentUser. The UID.
And if it's null, we're just going to pass anonymous. Import this.
Now let's write it to our Firebase Firestore instance collection of ad events.

And then I'm going to get the add function.

Let's make the function async.

The first thing I'm going to pass is the user ID. So the user ID.

Then we're going to have the ad type.

The ad type we've provided. And an event.

What sort of event has occurred?

Has the ad loaded? Or what?

A timestamp for when this happens.

Build value dot server timestamp.

And finally, unfold any extra parameters we possibly have.

Add a semicolon.

And then I'm going to set up the try cache block.

And the log event. Let's log it. Add event.

Fails to log which event it was.

Along with the error.

Add the missing dot here.

Now we're going to load our event. Replace this.

And I'm going to pass it the event first. Load. Build.

After that, I'm going to pass it the ad type, which is a banner.

Finally, the extra parameter is going to be a map, which is going to be an error code.

And pass it our errors code.

And then my error message.

Error dot message.

Now I'm going to add or dispose here since we won't be using the ad anymore.

And then I'm going to set the banner ad to null.

And finally, I'm just going to notify my listeners.

So we have a couple more events along with on load ad and on ad fail to load.

We also have the on ad opens.

You can customize this also for when a user taps on an ad and it opens an entire overlay of it.

But now I'm just going to add these logs because we don't need to dispose of the ad.

Replace this part. Opens.

We don't need the extra here because we're not logging any error.

And instead of fail, we're going to log open to Firebase.

Now if the user closes the ad, we're going to call on ads closed.

And then we're just going to log close here also.

After that, we have the impression.

On ad impression.

Replace this with banner ad recorded and impression.

Impression basically means the ad was viewed by the user.

And replace this with impression.

And finally, on ads clicked for where we don't want to open an entire overlay and instead

read out the user to another screen or in the browser. Was clicked.

And then I'm going to replace this with click. Okay.

So now our load ad function is completely set up.

Let's go ahead and save this.

Now in our show poll screen, I'm going to simply copy my speed dial from here. Control X.

And instead I'm going to wrap it with a consumer first of type poll provider.

And set up the builder.

From here, I'm going to first calculate a banner height variable and the poll provider.

Let's make the value of poll provider so it's clear. Provider.

So it's banner ads.

And if the ad is loaded, get the size.

And then the height.

And convert it to double.

And if it's not loaded, then just set to zero.

Because we only need padding from the bottom.

If our ad is not loaded yet.

To the padding, I'm going to assign edge and set struts only.

And to the bottom, the banner height. Semi colon.

And then I'm going to assign the child.

Now this is set up also.

This is going to dynamically add height between the banner and the floating action button

if the banner is loaded.

Finally, one last thing I'm going to do is inside my poll provider, I'm going to dispose

of my banner ad when we're closing.

I'm going to override void dispose function.

And then I'm going to get my banner ads.

And I'm just going to dispose of it if it's not already.

And then finally, super.dispose. Save this.

And let's go ahead and run our program.

Start debugging.

And now I'm going to go into my Firebase.

And here we don't yet have our collection.

Let me reload this.

And now since our banner has loaded, as you can see, our collection has been created as well with impressions.

Next I'm going to try tapping on it.

And as you can see, our clicked and opened are also being logged into Firebase successfully.

And our floating action button is also appearing correctly on top of the banner instead of overlaying on it.

In this lesson, we enhanced our implementation by logging banner ad events with Firebase, giving us better visibility into how our ads are performing and making it easier to monitor and troubleshoot any issues.

We also improved the user interface by repositioning the floating action button so it no longer overlaps with the banner advertisement, resulting in a cleaner and more user-friendly layout.

With these improvements in place, we have a solid foundation for integrating additional ad formats.

That's it for this video.

Thank you for watching and I'll see you in the next one.

4. Interstitial Ad

Hello everyone and welcome back.

In the previous lessons, we configured Google AdMob, integrated banner advertisements, and enhanced our implementation with Firebase.

In this lesson, we'll take it a step further by integrating interstitial ads into our Platter app.

Unlike banner ads, interstitial ads occupy the entire screen and are designed to be displayed at natural transition points within the app.

We'll learn how to create and load an interstitial ad.

Proper lifecycle management is important to ensure that ads are always available when

needed while maintaining a smooth user experience.

Let's get started.

So I'm going to start by creating a load interstitial ad function.

And then I'm going to call interstitialAds.load.

And then I'm going to pass it the required parameters, which is going to be the adsUnitId.

And I'm going to pass it the interstitialAdsUnitId.

Now let's go ahead and create this variable, like this one, opidisUnitId.

And paste the ID that I copied from the internet for the interstitial ad.

Now let's go back into our function.

And here I'm going to add a new parameter, the request, and set it to addRequest.

Finally, the addLoadCallback.

And pass it the interstitialAdsLoadCallback.

So on addLoaded, get the ad and create an interstitial ad and assign it the parameter run.

Now let's create this interstitial ads.

And then I'm also going to check if it is adIsReady.

Initially, it's going to be false, isInterstitialAdsReady.

Let's go back and get that here, interstitial ads.

Then I'm going to set isItReady to true.

And then I'm going to move on to my onAdFailed, pass it the error.

I'm going to load the ad event from here, get the loadFailed.

And then the ad type is obviously interstitial.

And finally, I'm going to pass it the extras parameter for logging our error, errorCodewordsError.code.

And then the error message was error.message.

And I'm going to set interstitial ads to null because it fails and isReady to false.

Now I'm going to create a new method to show our interstitial ads.

Void show interstitialAds, it's going to have the required void callback onAdClosed

and then also a build context.

Let's set it up.

I'm going to first check if the interstitial ad is not ready or null.

I'm going to log event, skip, ad type is obviously interstitial.

And finally, to the extra parameter, I'm going to pass a map.

Why are we skipping this?

We map the reason we are skipping this is ad not ready.

And then I'm going to call onAdsClosed.

And finally return it.

Next I'm going to set up an interstitial ad and get it full screen callback, assign it

the full screen callback contents.

First thing we're going to add here is onAdsShort, full screen contents, pass it the ads.

From here I'm just going to call loadAdEvents because it's loading successfully.

And here we're going to show shorts parameter and the type is interstitial.

Next I'm going to get the onAdsFails to show full screen contents, ad and also the error.

And we're also going to load an event here, showFails and the event type is interstitial.

And I'm going to copy this and add it here, add a comma.

After logging it, I'm going to add or dispose because we no longer need it.

Interstitial ad is going to be no, isItReady is going to be false.

And then I'm finally going to call onAdsClosed from here.

And then when the ad gets dismissed, onAdsDismissed, full screen contents, I'm going to load that

event into Firebase that it was dismissed.

And then the ad type is interstitial.

Finally I'm going to do add or dispose.

Then I'm going to get the interstitial ads and set it to no and the interstitial ready to false.

And if you want, you can also call loadInterstitialAds here.

So the next ad is already loaded.

If the user opens the screen again, and finally call the onAdsClosed here.

Now that's set up, I'm going to call interstitialAds.show.

Save this file and then I'm going to go into showPosts.

And on the init below my loadAds, I'm also going to load pollProvider.loadInterstitialAds.

So the ad is loaded on startup.

And then I'm going to go into the speed dial button for transitioning into a new poll screen.

And now since we have to add this interstitial ad at some navigation point, I'm going to

add it before a poll is created.

So every time a user creates a poll, we earn some money from it.

These ads can be very annoying for the user, so make sure to use them strategically.

And here I'm going to call context.reads, the poll provider, and then show the interstitial ad.

So the onAdsClosed function, let's create that.

Once the ad is closed, whether it fails or not, or it showed it successfully, navigate to our createPollScreen.

Now let's go ahead and save this and run our app.

It should begin loading successfully.

So let's go into our floating action button.

And from here, I'm going to go into the createPollScreen.

And as you can see, our banner ad is being loaded successfully.

And once the user dismisses it, it takes us naturally to the createPollScreen.

In this lesson, we learned how to load and display interstitial ads, respond to their lifecycle events, and properly dispose of and reload ads to ensure that they're always ready when needed.

We also discussed the importance of displaying interstitial ads only at appropriate transition points to provide a positive user experience.

At this point, your application supports both banner and interstitial ads, giving you a solid foundation for monetizing your Flutter apps with Google Ads.

That's it for this video.

Thank you for watching and I'll see you in the next one.

5. Integrating App Open Ads

Hello everyone and welcome back.

In this video, we're going to talk about app open ads, which are a great way to monetize your app by displaying a full-screen advertisement when users launch the app or return to it after it's been in the background.

Since these ads appear during natural transition points, they can generate additional revenue without significantly disrupting the user experience when implemented correctly.

We'll configure an app open ad unit and implement the logic to load the ad when the application starts.

Next, we'll display the ad whenever the app is opened or brought back to the program.

We'll also handle important life-cycle events such as loading a new ad after one has been

shown and gracefully handling situations where an ad fails to load.

So let's get started.

So the first thing I'm going to do is go into my features folder and I'm going to create a new folder here, called the ads folder.

And within that, I'm going to create a new file, app-open-ad-manager.

Because we'll be showing the ad on app load, we need a manager to load the ad and show it in the main class.

So I'm going to create a class, app-open-ad-manager.

And I'm going to create a variable, static constant string, and add unit ID.

I'm going to make it private to this class and I'm going to replace this with an open ad ID.

Next, I'm going to get an app-open-ad object imported.

And then I'm going to get a boolean, isShowingAd and set it to false.

Now that we have our variable set up, I'm going to create a load ad function here also.

And I'm going to get the app-open-ad.load.

And I'm going to pass it the ad unit ID.

The request is going to be an ad request.

And then finally, to the callback, app-open-load-callback.

Replace this, get our ad, and then assign it to our variable.

Then after that, we're going to go into our error.

And here I'm going to assign our ad variable null.

And now we need a logger for Firebase.

And from here, I'm going to copy this one and bring it over with me to our manager.

Let's paste this here and import all the necessary fields, also the log.

I'm going to rename this to logEvents.

And we don't necessarily need this field also, since these ads will be a lot.

So let's just remove this.

And then I'm going to log the events, load it, copy this, add it, load, field.

And then the extras map, error code, error.code.

And then the error message, error.message.

So now our load ad function is done.

Let's move on and create our void showAdsIfAvailable.

And it requires end-of-void callback.

On complete, import this.

I'm going to check if it's already showing it.

Then onComplete call, we don't need to show it again.

So we'll just call this and simply return from here.

Then I'm going to access my variable and get the full screen content callback, assign it a new object, full screen content callback.

Let's set up the ad shown on full screen content, get the ads.

And then I'm going to set isShowingAds to true and simply log my events that it was shown.

Then I'm going to get onAdFailed, set isShowingAds to false.

And finally get my log event, which is going to be showFailed.

And let's copy the extra.

After that, we're just going to dispose of our ads since it did not show and setAdOpened to null and call loadAd again so it can reload it.

And onComplete is finally called so it can navigate away from the screen.

Then I'm going to get onAdDismissed, get the ad, I'm going to set isShowingAds to false because user has dismissed it and then I'm going to log my events that the ad was dismissed.

And simply copy these again and add it here.

The loadAd is going to reload the ad for the next time when the user comes on the foreground of the app, it will already have an ad loaded up.

Then I'm going to get the impression ad and simply going to log an event from here that an impression was made.

Finally let's go ahead and appOpened.show our ad.

We'll also need to dispose of it, void dispose.

Now I'm going to go into my main class, if I save this, and then into my main.dots and above my main, I'm going to create a new appOpenManager, get the appOpenManager and assign it a new object.

Now below where I'm initializing my mobile ads, I'm going to await this first and then after that I'm going to call my app, appOpenAdManager.loadAd and this is basically going to load the ad when the app is initializing.

And then inside my init state, I'm going to add another observer, widgetBinding.instance.adObserver and pass it this context and we're going to add with widgetsBindingObserver.

Now we're doing this so we can add an observer to check whenever the app comes on the foreground,

we can immediately show our ad.

I'm going to create a new method, void didChangeAppLifeCycleState and pass it my AppLifeCycleState and now this

is going to check if the state has changed to AppLifeCycleState.Resumed which will only

happen when the app was backgrounded and now has come back to the foreground.

I'm going to use my appOpenManager to show our ads if it is available.

And then finally I'm going to create another and dispose, here I'm going to set the widgetBinding.instance

and I'm going to remove observer from here and then appOpenManager.dispose because we

no longer need it and super.dispose.

Just make sure we don't have any memory leaks.

Okay so we're done here, let's go ahead and save this, review, I'm going to remove this

extra import and our appID and our adID seems to be missing a little bit so let me re-paste this here.

We also need the format after the hash, make sure you're pasting the correct ID otherwise it won't work.

So let's save this and run our app.

And as you can see our ad is being loaded successfully on appstart, let's close this

and if I send my app to the background and then bring it to foreground again, a new ad

is going to load each time.

So in this video we successfully integrated appOpenAds into our Flutter application, we

learned how to load and display ads during app launches and when the app returns to the

program while also handling the ad lifecycle to ensure a smooth user experience.

That's it for this video, thank you for watching and I'll see you in the next one.

6. Controlling App Open Ad Frequency

Hello everyone and welcome back. In this video, we are going to improve our app Open Ads implementation

by adding a time interval between advertisements. Currently, our Open Ads can be displayed whenever

the app is launched or brought back to the foreground. However, showing an ad every time

the user returns to the app can become intrusive and negatively affect the user experience.

So to prevent this, we'll add a 20-minute interval between ads. We'll start by keeping

track of when the open ad was displayed. Then whenever the ad comes back to the foreground,

we'll check how much time has passed since the previous ad was shown. This gives us better

control over the frequency of our advertisements while still allowing us to monetize our app.

So let's get started. I'm going to start by creating a static variable for our duration

and it will be the minimum interval between shows and set its duration to 20 minutes. Next let's create a daytime variable. It will be the last shown time for when the ad was shown last. And then I'm going to create a function to get if it is too soon to show the ad. I'm going to access my last shown time and I'm going to check if it's now. Simply return false from here. Otherwise return daytime. So it's now. So a difference. And what the last shown time was. And check if it is less than the minimal interval between shows that we create. Next I'm going to create a new variable of duration type for checking the ad expiration date. And I'm going to set it to 4 hours. Now we need this because it's Google's policy that a loaded ad must be shown within 4 hours or it has to be discarded. Next I'm going to get the date time and the ad load and create a function to check if the ad is expired. Check the ad load time. If it is now simply return true from here. Otherwise return a date time. Do it now and check the difference. Ad load time is greater than or equal to our ad expiring duration. Next I'm going to create a variable to check if our ad is available that it has done loading. And then create a getter for it. Bool get is ads available. And create an arrow function. Either the ad is available. And. And it is not expired. Now in my load function before loading the ad I'm going to check if the ad is already available. Then simply return from here. No need to load another ad. Now inside our own ad loaded I'm going to set is ad available to true and set ad load time to date time dot now. And also inside my ad failed let's say the is ad available to false and the ads load time to null. Now inside my show ad available function I'm going to check if it is too soon to show a new ad because we're showing it with an interval of 20 minutes now. So I'm going to get the remaining time minimal interval date time dot now. And the difference. Get the last shown time. And then I'm going to get is minutes remaining dot in minutes. I'm going to call on complete from here. Now on complete of this I'm going to check if ad is available. Since it's not I'm going to load the ad from here. And call on complete dot call. And finally return from here. Now when the ad is opened I'm also going to set the last shown time to now. And from here I'm going to set it is available to false since it's already shown. Finally on add dismissal I'm going to add the same thing. So add it here. Now basics are set up. Let's dispose of it also. And this is it. Now when our app is running it will put an interval of 20 minutes before showing another ad. Previously whenever we opened the app it would show an ad on startup. And then if we opened our status bar or closed and reopened the app it would show an ad instantly. Now for a smoother user experience let's see how a 20 minute interval works. Now as you can see our first pop up has loaded. Let's continue. And if I close the app. Come forward or

open my status bar and come back to the app. It's not showing another ad. The next time it will show an ad is in 20 minutes. So our functionality is working correctly. In this video we improved our app open ads implementation by adding a 20 minute interval between advertisements. We tracked when the last ad was displayed and used the information to determine whether another ad should be shown when the user returns to the app. This prevents app open ads from appearing too frequently and helps create a balance between app monetization and user experience. That's it for this video. Thank you for watching and I'll see you in the next one.

14_riverpod-clean-architecture-and-http-networking

1. MVVM Architecture

Hello everyone and welcome back.

In this video, we're going to take a detailed look at MVVM architecture, which stands for module view and view module.

Before we start writing any code, it's important to understand the architecture we're going to use throughout this module.

The goal of MVVM is to separate the different responsibilities of our application so that our code is easier to understand, maintain, test, and scale.

Let's start with the three main components of MVVM, the module, view, and a view module.

The first component is the model.

It represents the data used by our application and is responsible for data-related operations.

For example, if our application needs to retrieve weather information from an API, the model can represent the weather data.

The second component is the view.

The view is what the user sees and interacts with.

In Flutter, this will usually be our screens and widgets.

The view should primarily be responsible for displaying information and responding to user interactions.

We want to avoid putting networking, business logic, and complicated state management directly inside our widgets.

The third component is the view module.

The view module acts as a bridge between the view and the model.

It contains the logic and state that the UI needs.

For example, if the user opens our weather screen, the view module can request the weather data, keep track of whether the data is loading, store the results, and expose the state of the view.

The view doesn't need to know how the data was retrieved.

It simply observes the state provided by the view module and displays the appropriate UI.

Let's take a practical example.

Imagine that we're building a weather application.

The user opens the weather screen.

The view displays the screen and asks the view module to load the data.

The view module then communicates with the appropriate data layer to retrieve the weather information.

Once the data is received, the view module updates its state.

The view observes the state and reveals itself to display the weather information to the user.

The important part here is that the view doesn't need to know how the API works.

It doesn't need to know which HTTP package we're using, what URL we're calling, or how

JSON is converted into Dart objects.

All of that responsibility is kept outside the UI.

This separation becomes especially useful as our application grows.

So why should we use MVVM in our projects?

Without an architecture like MVVM, it's easier for a fledged screen to become responsible

for everything from making API requests, parsing data, managing loading states, handling errors,

and displaying the UI.

As the application becomes larger, these widgets can become more difficult to understand and maintain.

With MVVM, we separate this responsibility into different layers.

The view focuses on the UI.

The view module focuses on UI state and application logic.

The module and data layer focus on retrieving and representing the data.

This separation also makes our application easier to understand and test because we can

test our logic independently from the UI.

It's also important to understand that MVVM isn't about creating folders just for the

sake of creating folders.

The real benefits come from separating responsibilities.

With every component having a clear purpose, we can change one part of the application

without having to rewrite everything.

So let's get started.

So here I've created this new project for a weather app.

And we basically just have our lib folder for now.

I'm going to start by creating a core folder, which will have our core functionality, for

example, the authentication, also our network data, any constants, such as our URLs.

These are some of the basic folders.

You can also add more functionalities in here like location or storage preferences.

Then inside my lib, a new folder for our modules, another one for storing our repositories,

for our view model, as we discussed, and our views.

And finally, a widgets folder.

So you can keep all the logic separate from the UI, the subsections of UI, our network

and core functionalities, and the repository that will be communicating with our bracket. And this is it.

This is the architecture you're going to be using for your large applications to keep

our business and UI logic separate.

So let's quickly recap what we covered in this video.

First, we explored the MVVM architecture and understood the responsibilities of its three

main components, which are the model, view, and the view model.

We learned how MVVM helps us separate different responsibilities within our application, including

the UI, state management, business logic, and data access.

Next we looked at how data flows between these different layers and how each layer communicates with the other.

We also discussed why using an architecture like MVVM is useful when building scalable

Flutter applications.

As our application grows, keeping these responsibilities separate makes our code easier to maintain,

test, and extend.

That's it for this video, thank you for watching, and I'll see you in the next one.

2. Riverpod Introduction

Hello everyone and welcome back.

In this video, we're going to start building our weather app using Riverport and the MVVM

architecture we've already set up.

So let's go ahead and get into what Riverport is.

Riverport is a state management and dependency injection solution for Flutter applications.

It allows us to create providers that can expose and manage different types of data,

services, and application states.

One of the main benefits of Riverport is that it helps us keep our application logic separate from IoI.

Riverport also provides compilers and safety, makes our code easier to test, and scales

well as our application becomes more complex.

This makes it a good fit for the MVVM architecture we'll be using in this module.

So why use Riverport?

Now that we know what Riverport is, let's look at why we should choose it for our applications.

One of the main benefits of Riverport is that it's reactive state management.

When our application state changes, the UI can automatically respond to those changes.

Riverport also provides dependency injection, which will be particularly useful in our MVVM architecture.

We can use providers to make things like repositories, services, and view models available wherever they need it.

Riverport also provides strong type safety and works well as our application grows in complexity.

Now let's briefly compare Riverport with some of the more traditional approaches to

state management.

With traditional approaches, it's common to manage states directly in styled widgets

or rely heavily on build context to access and update application states.

As an application grows, this can make our UI widgets more tightly coupled to our business

logic and dependencies.

Riverport gives us a more structured approach, our state and dependencies can live outside

of the widgets, and providers can be used to expose the parts of the application that need them.

Another advantage of Riverport is that it doesn't require us to depend on build context

to access our providers.

This can make our code easier to organize and test, especially when we're working with

an architecture like MVVM.

So let's get started.

I'm going to start by creating a new repository, repository.path, and create a new class here.

First thing we're going to create here is an API client.

It is primarily going to be responsible for sending our GET and POST requests.

And let's create and write the repository and pass this dot API client.

Now let's go ahead and create the client in our networks folder, API client dot Dart,

and create a new class.

I'm going to create a future of HTTP dot response, import that, and then create the GET method.

We're going to provide it a string URL each time for which API we want to hit.

Finally, I'm going to get the URI from URI.path, get its response, await, HTTP dot GET URI,

and then finally return the response from here, make it async.

Now this is just going to be a sort of wrapper that we use each time we want to make a GET request.

So we don't have to repeat these three lines in every request each time.

So let's go back and import it here.

Let's go ahead and create a method for fetching our weather input, future of void, and fetch weather.

It's going to get the latest weather data based on the location that you want.

I'm going to get the URL, which is going to be the same one each time.

So for that, you're going to have to go to visualquestion.com, which provides a free

API key that you can use for development purposes.

All you have to do is get your free API key and sign up.

Since I already have a key, I'm just going to input my account data.

And I already have my API key.

Let's go to the weather query builder and instead go to the docs and the weather API.

And from here, you can copy this base URL endpoint, which we're going to use.

Let's copy this.

And we're going to fetch our weather information for that.

So I'm going to create a constants class, API constants, both that, you create a class

for this, the API constants, and I'm going to create a static string for our weather,

base URL, and passage the URL.

So now that we have our free API, let's also set the weather API key, static constants,

string, weather, API key, and paste the one you already copied.

Let's save this.

I'm going to access the API constants here, import it, dot, our base URL, then attach

our query, which is going to be URI dot encode a component, which is going to be our location.

And we're encoding this because locations tend to have special characters, and we don't

want the URL to be messed up.

Also attach the key here, using our API constants, dot, weather API key.

Now that we have our URL set up, get the response here, using our API clients, dot get, using

our URL, and check if the response, dot status code, is 200, in which case it was successful.

I'm going to try parsing it here, because we don't want to be parsing it anywhere else.

So get the JSON, and decode it.

So this is going to be the response, dot body, and return it as a map of string and dynamic.

Let's return it from here, weather, dot from json, and pass it the JSON object.

And I've already created this weather module inside my models, because it's going to take

a long time to type this up.

I'm just going to explain how these models work.

Basically we have our main weather border, it's going to have our resolved address, the time zone, address, and then it's going to have a list of weather days.

So let's go ahead and do that.

Weather day is going to have multiple days.

You can get up to 16 days worth of data, in the free tier, from the Visual Crossing API.

So it's going to have the date time, the temperature, what it feels like, the humidity, and other icon.

It's going to be an online URL to the icon.

And the sunrise and sunset time, other than that, it's going to have our precipitation type, and then our weather hours.

And finally, within the weather hour, we're going to have our date time, in that hour, what the temperature is going to be like in that hour, what the humidity will be, and what the conditions will be. So that's it.

Those are our three models.

Let me import the weather model here, and change the return type to weather.

Since it's possible that they could change their format at any time, you should check that on format exception, that it was not correctly parsed into a JSON, we want to throw an exception, failed to parse weather JSON.

Otherwise, if the status code is not 200, just copy this, and add it here.

Next, we're going to fail to fetch weather, and replace this with a response.status code.

So now our weather repository is also set up, let's save this, and close the other files.

Next, I'm going to create a new model, weather, view, model.dart.

And I'm going to get a final API client provider.

And for that, I'm going to use Riverford provider of type API clients, and then get its reference from a new API client.

If it exists, then you use that, otherwise create a new one, import the provider from our Riverport.

Now this is basically going to create a Riverport provider that gives us one shared API client.

Next I'm going to create a weather repository provider from a Riverport provider, which type is going to be weather repository that we've already created.

And I'm going to pass it the reference, reference of our weather repository instance, and it's going to read the API client provider.

The read in our Riverport means fetch the value once, like in Firebase, if we get the log once we use .get, and in Riverport we have .read.

If you want a stream of the value that whenever it updates, it should update the UI automatically, you should use .watch here instead.

Next I'm going to create a weather, view, module, provider, and async notifier provider of the types weather, view, module, which we'll create in a second, and also a weather instance and get the weather, view, module, .new.

Now this async notifier provider, it's a Riverport provider that owns the async state.

This model class is going to be a class that holds the actual state, weather is going to be the type, and the new is going to create a new instance when needed.

Now let's go ahead and create it.

I want to create a class of weather, view, module, which is going to extend an async notifier of the type weather.

Now let's create the build method for this, override future of a weather and the build.

From here I'm going to return the result of a function, and from here just provide a random location for now, such as New York.

Now I'm going to create the fetch method, and I'm going to delegate the actual call to our fetch method.

It's going to have the location, and return the reference, .read, read the weather repository provider that we have, and fetch weather from it using our location.

Replace our return type.

Now our models, repository, and weather view model are all set up.

Let's go into views, and create a new view from here, a weather screen, dot dot.

Let's put it inside a weather folder instead of directly putting it inside the views. Move it.

Now I'm going to create a stateful widget for the weather screen, and I'm going to replace the stateful with a consumer stateful widget for our river pool.

And also replace this normal state with a consumer state.

Copy this, and replace this state also.

So this is how you set up your river port screen.

Instead of a stateful widget, you're going to have a consumer stateful widget.

And now inside my build method, I'm going to call my weather state, and get the reference to watch my weather view model provider.

Now as you can see, I'm using watch here.

So whenever the data is updated, I want my UI to be updated out.

So let's go inside our main.dart, and here, I'm going to replace run app with a provider

scope, import it, and then pass it our child, which is my app for now.

And now I'm going to replace my home screen with the weather screen that we created.

Remove the boilerplate code, and our main.dart is set up also.

So our river port flow starts from our provider scope.

When we call our weather view model provider, it comes in here and instantiates a new provider

using the async notifier provider that we have, which goes into the build function,

and calls our fetch function for fetching the location.

And then it reads the value from our fetch weather to which we pass the location.

So that's the basic flow for a river port provider for fetching the information from a repository.

So let's recap what we covered in this video.

First, we introduced river port and discussed how it helps us manage state and dependencies

in our Flutter application.

We also looked at some of the benefits river port provides and how it fits naturally into

the MVVM architecture we'll be using in this module.

After that, we started setting up our weather application.

We then created a basic weather repository, which will be responsible for communicating

with our weather API and fetching the weather data.

Instead of making API calls directly from our UI, we've kept the responsibility inside the repository.

This helps us maintain a clear separation between our UI and the data fetching logic.

We also write with the basic weather models that we'll use to represent and work with

the data returned by our API.

So that's it for this video.

Thank you for watching and I'll see you in the next one.

3. Building the Weather Screen Part-I

Hello everyone and welcome back.

In this video, we're going to start working on the weather screen of our application.

So far, we've created our weather modules and repository, and we've implemented the

logic required to hedge weather data from our API.

Now it's time to bring that data into our UI and actually display it to the user.

We'll also start learning how we consume this data using Riverport and Connector UI with the state managed by the application.

So let's get started.

So let's get started on our weather screen.

I've already created a couple of gradients for our screen since weather apps tend to have a lot of gradients.

I set this up according to the icon we'll get from our weather API.

If it's a clear or day, we're going to use a certain kind of gradient for showing a normal sunny day and the same for clear and rain and thunder, snow, etc.

So these are going to be various gradients that we use depending on the state of weather we have.

Next, we also set up the linear gradient for our default while the icon is being fetched from the API.

And then the icon emoji, it's going to be processing what kind of icon we show depending on the text in the icon that we have.

So let's get started on the weather screen.

I'm going to create a text editing controller here for our search bar.

So replace this with just controller for now.

I'm going to replace the placeholder with the scaffolds widget, turn the background color to colors draw transparent, import the UI materials library.

And then I'm going to set the emoji to an animated container.

So we can animate the flow of weather.

I'm going to set the duration to milliseconds, 800 and set the curve to curves dot ease in out.

Next, we're going to need to set the decoration and for the decoration, I'm going to set it to box decoration and we're going to get the gradient from our UI state instead of calculating it in the UI.

So for that, we're also going to need our providers.

Let's create this UI state.

I'm going to create a UI state rep dot watch a weather screen provide.

So to manage my UI, I'm going to create a new view model and I'm going to call it the weather screen state dot dot.

Now here I'm going to create a class for my weather screen states.

I'm going to create a string for my validation message.

If there is an error loading from API, we show it.

Then I'm going to get the linear gradients.

The current gradient will be using then create the const weather screen state and I'm going to pass it this dot validation message and set it to empty by default.

And then this dot current gradients and today I'm going to set the K default gradient.

Let's also create a copy with method for this weather screen state, get the validation message and then a linear gradient, the current gradient.

Now it's going to return a weather screen state and assign it the validation message that we have.

Otherwise the default one we have set already and then the current gradient, this dot current gradient.

Now I'm going to create a notifier for this.

So a class of weather screen notifier extends our notifier of type weather screen state

import riverport.

First I'm going to get the build override weather screen state and then get the build method and then get the weather screen state.

Now I'm going to create a validation message, get the message and then set this state to state dot copy with, keep the gradient, but update the validation message.

Let's do the same for our gradient, set gradient, a linear gradient that we have and set it to GR current gradient while keeping the message same.

Now I'm going to create a provider for this so you can access it, weather screen provider and get the notifier provider, pass it our weather screen notifier and also our weather screen state and create a new instance of weather screen notifier dot new.

So now this is set up, our screen state will be managed by this class that we created from our validation message to the gradient that we'll be using.

Let's go back to our weather screen.

And now whenever the build is rebuilt, we'll also be rechecking what to rebuild our UI.

Let's import this.

Now I'm going to set the child to a safe area and then a column, set the children, here the first thing will be our top bar.

So I'm going to use a private widget for this.

So let's create it, widget build top bar.

I'm going to return a padding of edge insets, set it to 16, 12, 16 and 4.

Next I'm going to set the child to a row widget, set its children, first I'm going to use an

expanded widget and then a container.

Let me quickly set the decorations for this container, book decoration, color, I'm going to set to colors dot white.

I'm going to set the alpha to 0.2, then the border radius, and next get the border, border colors dot white dot with values.

Now to the child of the container, I'm going to assign a text fields for our search bar, set the controller that we created, let's move this inside our stateful widget.

I'm going to start styling it next, set the text style, color is going to be colors dot white, and then quickly set up the decorations, set up its input decoration, set the end text to search city, so you can search any kind of city, then set the end style, text style, color will be colors dot white 60, so it doesn't blend in with the background color, get a prefix icon, icon button, and we'll be performing the search on this tab later.

For now let's set the icon to icon, icon start search, and then set its color to colors dot white 70.

I'm also going to set up a suffix icon for refresh, so user can click on that to get the latest weather data, refresh, round it, and keep the same color.

Next set the border, but let's set it to input border dot none, next the content padding, so the inner text doesn't touch the edges, and set it to symmetric, horizontal is 20, and vertical is 40.

Let's move on to our text input action, what is the action we'll be performing is input action dot search, and finally unsubmit it, I'm going to set it to nothing for now.

Now let's build the body of our column, I'm going to use an expanded widget, I'm going to be assigning a child weather, and I'm going to be assigning it a child depending on the weather state, so when it changes, data is going to be our weather data, get weather content, we'll be creating this widget, next on error, get the error, and build a widget for that, pass it, next create a, while the data is still loading, so build loading widget, now we have to create these three kinds of widget, let's create the shells first, the class, let's create the build loading first, loading, it doesn't have any parameters, so I'm just going to return a center widget, and set the child to a column, I'm going to set its main axis size to main axis size, minimum, and then the children, show a circular progress indicator, and set the color to colors dot white, because by default we'll have the blue gradients, and then set the size box for a little bit of spacing, set it to 16, and then also show a text widget, weather, and let's set the style to text style, color is going to be

colors dot white 70, and then font size will be 16, let's replace this class with a widget, so our build loading is done, next let's create our build error widget, the lower loading, build error, it's gonna get an object, error, and return a center widget, padding, all 32, now set its styles to a column widget, its main axis size is gonna be minimum, set the children, first I'm gonna get an icon to display icons dot cloud of rounded, that we were unable to fetch the info, and then color is gonna be colors dot white 54, size will be 72, next create a size box with a height of 16, get a text to display our error, dot to string whatever message we have, set the style to text style, first the color will be colors dot white 70, font size will be 14, and finally text alignment is gonna be center, copy the size box and edit the lower text, and increase it a bit, next I'm gonna get an outline button, outline button dot icon, for now I'm going to set it to nothing, and add a label for a text, try again, and set its style to text style, set the color to colors dot white, next I'm going to set the style of the button itself, outline button, outline button dot style from, set the sides to border side, and the color will be a bit darker, so colors dot white 54, and then set the padding, edge inserts symmetric, horizontal is 24, and vertical is 12, so our build error is done, let's go ahead and create our weather content, going to create it outside my weather screen, let's create a stateless widget first, and I'm going to replace it with a consumer widget, and pass it the widget reference, widget ref the reference, now it's going to be needing a final weather object, import this, and replace it with the final parameter we need, and for now I'm simply going to replace this with a text widget, and then I'm going to access weather dot days of zero dot, let's get the fields like for now, and convert it to two string to display the temperature, and before displaying it we should check if weather dot days dot is empty, then simply return a center widget, set the tiles to a text that no weather data available, and set the style to textile, and the color to colors dot white 70, and font size to 60, now we need to pass it the weather from here, so let's pass it the weather, and let's see what other error we have from our debug console, the problems, and simply add a semicolon, now let's save this, let's run our program, so as you can see our temperature is being displayed correctly at 79.1 degree Fahrenheit, in the next video we'll start working on setting up the main widgets inside the weather screen, so we can properly display our weather UI data, so in this video we started building our weather screen and used it to display the data we retrieved from our API, we connected our UI to the application data layer using riverport, and learned how our widget can consume the state provided by our view module, we also looked at how we can handle different states such as loading, success, and error, so that our UI can respond appropriately depending on what's happening with the data,

so that's it for this video, thank you for watching, and I'll see you in the next one.

4. Building the Weather Screen Part-II

Hello everyone and welcome back.

In this video, now that we have our basic weather screen connected to our API data,

let's continue building out the user interface.

In this part, we'll add more weather information to the screen and improve how that information

is presented to the users.

We'll work with the data we're already receiving from our ViewModel and use it to

build different sections of our weather screen.

As we add these UI components, we'll be using Riverport to access the application

state rather than handling the data directly inside a widget.

Let's get started.

I'm going to start by accessing the value of today here.

So weather dot days of zero.

So today is going to be the first value it has.

And I'm going to replace this text with a single child's value.

I'm going to set the physics to bounce physics.

And finally to the child, I'm going to assign a column.

I'm going to set the cross-axis alignment to start, so cross-axis alignment dot start.

And then I'm going to create a widget here, set it to the hero section.

And we're going to be passing it our weather data, also a single day.

So the data of today.

Let's create it.

Let's create a state-based widget, call it the hero section, remove this key.

Next I'm going to create the weather.

And then also my weather day data for today.

Let's import that.

And add these parameters.

I'm going to replace this placeholder with a padding widget, set it to edge insets dot

symmetry and set the horizontal to 24.

Next to each child, I'm going to assign a column and children.

Set the size box to a height of 12.

And then a row for main axis alignment, main axis alignment dot center.

And then to the children, let's first set an icon for icon store location on, and set

its color to color store twice, 70, and size to 10.

Then another size box here for a width of 4, since it's a row, and then a flexible for a text widget.

And to the data, I'm going to assign the resolved address, whichever address is showing data code.

Set its style, text style, color, color store twice, and then font size to 16.

And finally, if the data is too much for the address, set the text overflow.

Let's make these required.

And replace this with a column.

Now below my row, I'm going to add another size box, delta height to 28.

And then the text.

I'm going to be setting this to an icon emoji, show the emoji we get from the ADI.

And then the style is going to be text style.

I'm simply going to set its font size to 90.

So the icon is bigger than the texture.

Finally another size box for a height of 8.

And then get a text.

Here I'm going to show data of today dot temperature.

Show the temperature as A2 fixed, two string as fixed, and then set it to zero.

So the first data and change the data we're getting is in Fahrenheit.

So degree Fahrenheit.

Let's set the style, text style, color is going to be color store twice, font size is

going to be 88, font width is font width dot W200, and height is going to be 1.

Below that, add a new size box for a height of 4.

And then get the text, we show today dot condition, what conditions we have today.

And let's copy the style from here, add it.

Let's make this white 70, it's going to be 20, and set the font width to 300.

Let's get the size box and height 8.

Main axis alignment is going to be center, and set the children.

Next I'm going to get the temps badge, and set the label to high, for that I'm going

to use H. And then the temperature is going to be today dot temps maximum.

So the higher limit, and if it's hot, true.

Now let's create this temp badge, create the state page widget, temp badge.

I'm going to add a final string label here first, as a ready hint, then get a temperature of double, final, blue is going to be hot.

Add these to the parameter, and let's make this required, add required for each one.

I'm also going to set its padding, H, and set dot symmetry, set the horizontal to 12, and the vertical will be 4, horizontal, now set the decoration to box decorations, and the color is going to be colors dot white, dot width values, and alpha is going to be 0.15.

Border radius is border radius dot circular, enter.

Next I'm going to set the child to a text widget, to show first the label, and after that, I'm going to show the temperature, to string, as speaks, degree symbol we've added here, let's copy this, and add it next to our string.

Next I'm going to set the style to textile, color is going to be depending on whether it's hot or not, so colors dot orange if it is, shade of 200, otherwise colors dot light blue for a normal scenario, then get the font weight, or font weight dot w600, and then the font size will be put.

So I've added our high temperature here, let's also add for a low one, so let's copy this, I'm going to add a size box, between this, the height, width of 12, and add a low badge, set this to false, and temperature minimum.

Now below my low, I'm going to add another size box, height of 12, and a padding widget, as in sets, dot symmetry, and horizontal is going to be 60.

To the child, let's assign a text widget, weather dot description style is going to be textile, and color is going to be colors dot white, 60, and then the font size will be 30.

Also add a text align, to make the text align centered.

So now our hero section is done also, let's go back to our weather content, and below this I'm going to add another widget, after my size box, so set the height to 24, and then get a stats row, which will be showing our data for today, so today.

Now let's go ahead and create the stats row, create a stateless widget, for stats, show, and then add the final, weather, day, and get today's data, remove this, and use prompt.

This is the placeholder with the padding widget, set as in sets, dot symmetry, horizontal, horizontal will be 16, close it, and set the child to a row, the row is going to have children of the type, this type, so let's create that first, another stateless widget, of type stats,

tile, remove this parameter, get the icon data for each tile, so icon, input, then get the string, value, and finally the string label, add this, and let's make this required, let's replace our placeholder, with an expanded widget, so the row is fallible, with this I'm going to first assign a container, let's set the decorations, horizontal will be 4, then the padding, vertical will be 14, and horizontal will be 6, then set up the decorations, box decoration, color is going to be colors dot wise, with values, set the alpha to 0.12, gorgeous, so now we're done with the decoration, let's move on to the child, going to assign it the column, set the main exercise, to main, see it's size dot minimum, and then the children, let's assign it an icon that we have, icon, and then set it's color, color is dot wise 70, and then the size is going to be 22, set the size of 6, and then a text, with the value we have, and set the style to text style, color is going to be colors dot wise, font width is font width dot bold, and font size is cortex, next set the size box, with height 2, then another text with the labels, and then the style, text style, color is going to be colors dot wise 60, and font size instance, and set the text align, so the text is aligned to send, now let's add this text styles, let's add the first icon to thermos set, the value is going to be today dot tails like, dot to string as fixed, and zero, add it here, remove the label, it's going to be tails like, because we're showing tails like that, and instead of thermostat, I'm going to show a water drop, and instead of tails like, let's show humidity, and replace this with humidity, another one for showing wind mile per hour, and today's wind speed, and get the air, finally show another one for rain, replace with umbrella, and it feels like we've perceived, so let's run our program. So, as you can see, our icon and other data is showing successfully, along with our location, our temperature, and other stats that we showed below our temperature. All of this data is being fetched from the API, and all we did was use the report and MVVM to show this and create an entire app. So, in this video, we added more weather information to the UI and worked on presenting the data in a more useful and organized way. We also connected additional UI components to the state managed by Liverpool, allowing our screen to display the data dynamically instead of relying on hard-coded values. We now have a more complete and reactive weather screen, and we'll continue building on this foundation in the upcoming videos. So, that's it for this video. Thank you for watching, and I'll see you in the next one.