BLoC Zero to Hero — State Management

Transkrip referensi — folder sumber `2_flutter_bloc-zero-to-hero`

11 bagian · dibuat otomatis

Daftar Isi

1_why-bloc-as-a-state-management

1. Why BLoC?

Hey, what is going on, everyone?

And let me wish you a huge and warm welcome to the first chapter of this amazing Flutter block course.

Today, I'll be guiding you throughout an introduction on Block, explaining why I chose Block as my favorite state management library and design pattern going through the main aspects of it.

And hopefully at the end of the video, you'll have yourself some good reference on why it's an amazing idea to start and use block on every project from now on.

So without further ado, let's get right into it.

But first of all, let me ask you a question.

Have you ever installed an application which started to crash for no reason while using it?

An app perhaps that after you clicked a button in order to load some data freeze for three four seconds without displaying absolutely any message into the UI, or maybe an app which only works with internet connection but goes haywire if there is no one available.

I'm assuming that at least for some of you, the answer will be definitely yes.

Then you will know how frustrating it is for a user to not receive a visual representation for any interaction with the application.

The main problem here is that the code is not clear, organized, maintainable nor testable.

Developers of these apps are thinking about what works for the moment instead of reflecting what would work in maintaining the app in the long run.

The main idea in programming is that you and your team need to write code using an architecture which fits you the best an architecture which can accommodate every feature and scale over time.

Now let me give you a quick story.

In my first weeks of learning Flutter and Dart from some tutorials online, I wanted to give it a try and build my own first app by myself.

But eventually, after a couple of days of coding, I was struck by the feeling that I cannot maintain the code anymore due to the fact that all of it was spread throughout the app randomly.

So there was the moment when I learned the most important lesson.

I have started to look for a way to organize and structure the ideas and features in my code in order

to build an application that will be maintainable and scalable over time.

In Flutter.

The best way to keep your code organized, clean and maintainable is to have a component which can mediate

what the user sees, often referred as UI, front end or presentation and the logic behind it.

This components I'm referring to is, as you may have already guessed, Block, which is an abbreviation

for business logic component.

Generally speaking, Block is a design pattern created by Google to help separate business logic from

the presentation layer and enable developers to reuse code more easily.

There is a state management library created and maintained by Felix Angelo called Block, which helps

developers implement the block design pattern in their flutter applications.

What does this mean actually?

Well, as developers, it's really necessary to know what the state of our app is at any time.

Remember when I said that for every interaction with the app, there should be something displayed on

the screen to let the user know what's going on.

Technically, for every interaction, the application should be in a different state.

For example, when the data is fetching, the app should be in a loading state, displaying a loading

animation on the screen.

When the internet is off, the application should display a pop up to let the user know that there is

no internet connection.

This is good app design thinking of every possible case and having a state emerging from it.

So not only does the Block library separate presentation from business logic, it also simplifies the

way we manage the state of the application.

So after all, by learning Block and using the Block library, you'll have the following huge advantages

which will lead you and your team to be better developers.

Block will allow you to have a better understanding of what's happening inside your own app, thus making

it more organized, structured and easier to test and maintain.

There won't be a moment in which you won't know what the state of your app is because of the way Block

is built.

For every user interaction, there will be an emerging state making the whole state management issue

simple to overcome.

And all within a single, stable and powerful block code base.

In the next chapter, we'll start taking a look at every block concept like streams, blocks and qubits while also discussing the difference between them is going to be a really, really important one.

So make.

Sure you won't skip it?

With that being said, I'm looking forward to seeing each and every one of you up next.

Stay tuned and let's get Wicked.

2_bloc-core-concepts

1. BLoC Core Concepts - cubits, blocs, events, methods, streams

Hello, everyone, and welcome to the second chapter of this amazing course in which we're going to dive into the core concepts of Block.

And I have to tell you that it is critical to understand them before trying to use.

Block So mark this as an important lesson because you may need to revisit it multiple times in order to understand every concept.

So without further ado, let's get right into it.

The first major concept you need to get familiar with is the concept of a stream.

This is the foundation of block, so please pay as much attention as possible because I will try to make it as simple as possible to understand.

Imagine the following scenery.

You are with another friend in a beautiful mountainous area and you spotted a nice river flowing from top to bottom.

You decide to take part in a recreational activity.

You will go uphill to where the river starts flowing and your friend will wait downhill.

You are not able to see each other since the river swimmers left and right multiple times.

Now the task of this activity is that you will send no more than ten boats down the river containing multiple numbers inside them, corresponding to the order they were dispatched.

The first boat will have number one, the second will have number two and so on and so forth.

The purpose of this activity is for your friend to shout out the number he will receive at the end of the river.

So let's simulate this activity for a bit.

You will send the first boat with the number one inside.

Your friend is waiting patiently for it to arrive downhill.

It eventually arrives after a couple of moments and your friend shout outs one after two seconds, you send another boat, your friend receives it and shout outs, two and so on and so forth until there are no more boats to arrive.

If you understood this little activity, then you understood what a stream is.

A stream is at its foundation, mostly a river which transports some data on a boat from the sender to the receiver.

The transported data is asynchronous, which means the receiver doesn't really know when he'll receive the data.

He just waits or listens to it.

Now let me attach the code to every component of this entire recreational activity.

Just take a look and observe how natural everything goes into place.

So there is a boat stream function which returns a stream type of integer numbers.

This will be a function designed by you, the guy who is sending all the boats down the river.

In this function, all you are doing is iterating ten times in a for loop in order to send the boat with a corresponding number.

You may notice that there are three keywords that we have not discussed yet.

Async star await and yield.

I told you before that the data or the boats in our case that are being dispatched are asynchronous.

Well, the function from where all the magic happens needs to be an async generator function because it generates asynchronous data, hence marking it with the async star keyboard.

In an async star function, the keyword yield pushes the boat through the Stream river.

So every time we yield something in an async star function, we are sending that piece of data to the stream.

The await keyword helps waiting for a process to finish before proceeding to the next step.

In this case, we are waiting for a two seconds gap between boat dispatching.

Okay, so we took care of the guy who sends the data.

But what about the guy who receives it?

Again, we'll notice how natural the code sounds like in the main function.

All there is to be done is initializing the stream.

Then we will listen to the stream by analyzing whether a boat is approaching or not.

Remember, your friend doesn't know when the boat will arrive down the stream, so he needs to listen to it or practically watch for it.

But eventually you'll ask yourself a question Why streams?

Why do we need them?

You might remember from the last video on why Block When I mentioned that for every interaction inside the app, there should be a state emerging from it, right?

Can you see where streams will come in extremely handy in this situation?

So you will have interactions emerging from the app which needs to be listened to in order to emit state based on the sequence of code which changes the app.

Take a look at both of these next concepts so that everything will be more clear, both the video and have a couple of seconds to properly read them and try to understand what's going on.

Both qubits and blocks can also communicate with the outer.

A data layer.

But this part is beyond the purpose of this video and will get covered later on.

So let's remove this part for now.

What can we observe here?

Qubit and block are two really important concepts which you need to understand.

The only difference you should have in mind for now is that a qubit is a minimal version of a block, so Block actually extends cubic.

To understand why is that we need to dive into what is a qubit and what is a block.

Let's start with a qubit first.

A qubit, as you can see from the picture above, is a special kind of a stream component, which is based on some functions which are called from the UI functions that rebuild the UI by emitting different states on a stream to make the understanding process simpler.

Let's associate the concepts with the default counter app from Flutter.

So question number one, what is the initial state of the app when you start it?

In this case, the default counter value defaults as zero.

Question number two what is the main interaction the user can have with the app?

Well, a user can click one of those plus and minus buttons.

And question number three, what are the functions which need to be listed into the qubit component in order to describe the actions defined by the minus and plus buttons, we'll have two functions, one called increment and the other called decrement, which will logically add or subtract one to the current state of the counter.

Now let me show you the implementation of the qubit to the screen so that you'll completely understand every question and answer from above.

Here we are declaring the qubit responsible for managing the counter feature of the app.

As you can see, we are setting the default state of the counter to zero right into the constructor.

Then we are creating an increment and decrement function so that when we call it from the UI, the new

incremented or Decremented state will be emitted.

The main function acts like the UI currently, so don't worry if you don't see any plus or minus buttons to play with right now.

The function was just designed for educational purposes and ease of understanding.

We'll get to flutter user interface and widgets in the next tutorial.

So inside the main function we are declaring the counter qubit.

Then we can observe how we can update the state of the counter value easily by calling the specific functions we created earlier.

And since the qubit is actually a type of stream emitting state, you can listen to it and observe the state changing.

Take in mind that, as you can see, compared to block, the qubit functions are not part of a stream, but rather of a predefined standalone list of what can be done inside the qubit.

Actually, the important difference is highlighted by the lack of these blue points on the schema.

The one and only stream existing in a qubit is the stream of emitted states.

So you may now ask what advantages could block bring into the game?

Isn't qubit enough?

Remember that qubit is a component which, based on some functions, I repeat, functions that are not part of a stream emits a stream of states which modify the UI.

Block, on the other hand, is more advanced since not only emits a stream of states but also receives a stream of events.

So think of block as being the brain of an advanced and complex component from your app.

Depending on the interactions the user has with the app, there will be an event emerging from it which comes as an input to the block.

Then inside the block there will be a required mapped event to state function, which will take the event and convert it into a state so that the UI can successfully rebuild.

Let's take a look at the same counter app example and turn the previous qubit into a block.

So here, as you can see, instead of having a pre-cooked increment function, we have an enum called Counter Event which contains two events the increment and decrement ones which can be entered as an input into the map event, state functions and then a new incremented or decremented state will be yielded

into the output state stream.

As with the qubit, in order to use block, you have to declare it whenever there should be a new event

dispatched.

We call block, dot Add and the event we want to.

For example, when the plus or minus button is pressed, there should be an increment or decrement event dispatched.

The block brains will listen to the event stream, receive the event.

Just how your friend was receiving the boat on the river actually.

Then then calculate the next value and yield it on the state stream.

Now that we've covered the basics of the qubit and block classes, you might be wondering in the first place when you should create a qubit or a block in your app.

And in the second place when you should use qubit as opposed to when you should use Block And what is

actually the real big difference setting them apart from each other?

So let's get right into question number one.

When should you create a qubit or a block in your app?

This is a really important question you will ask yourself when you finally understand the concepts of these two.

Technically, there should be a block slash qubit component for every feature of your app, and by feature you should understand a key functionality your app offers.

For example, you have a customer tab where you can create, show, delete some people from a list.

Then you will need either a qubit or a block to manage the events of creating, deleting, showing a customer and the states emerging from them which will change the UI accordingly.

Another example can be the type of internet connection you have.

You will need a block or a qubit to keep track of what is going on with the app when there is no internet connection or when there is a poor one or when connected to Wi-Fi rather than mobile data and so on and so forth.

So I hope it's clear for now.

And now here comes question number two.

Where should you use qubit as opposed to block?

And what is the real big difference setting them apart.

So as I have previously said, the main major difference between a qubit and a block is related to how each one of them receives input from the UI.

So you may ask why this difference?

Why is there a qubit in the first place?

Isn't block an already amazing and powerful solution for everything?

Well, technically it is.

You can use block as opposed to qubit everywhere in your app.

You don't even need to care about qubit if you ask me.

But you see, there's no real need for a stream of events when you want to implement a simple counter feature using a stream for events to do.

This is like getting a bulldozer to dig in a 50cm hole in the ground where you can simply use a shovel to do it.

So when do stream advantages come into place then?

Well, think about searching for a weather location, for example.

You are typing each letter and for each one there is a new list of locations displayed from where you can choose from.

What would a qubit do in this situation?

Let's think a little.

It will call the get locations list function.

For example, for each letter you type into the input field.

What would a block do?

Well, you can set the block to call the get locations list function once every four letters, for example, or maybe once every two seconds, or maybe when the user stops typing.

And that's because the input events can be delayed and manipulated thanks to them being dispatched on an actual stream.

Okay.

So having that in mind, we can deduce that block is the big brain of the project, whereas qubit is a smaller part of it which can be used to optimize code workflow.

The advice I can give you right now is to start with a qubit and modify the qubit into a block when you really, really need the extra stream juice.

Having all of this in mind, in the next chapter, we'll discuss another really important topic, the flutter block concepts.

Maybe you heard about block provider, block builder and block listener along the way.

These are some of the flutter block concepts we'll tackle up next.

With that being said, I'm looking forward to seeing each and every one of you up next.

Stay tuned and let's get Wicked.

3_flutter-bloc-concepts

1. Flutter BLoC Concepts

Hello, everyone, and welcome back to the third chapter of this course.

Remember the concepts we discussed in the previous chapter?

The streams qubits and blocks because we will use them as the foundation when learning the new flutter block concepts.

There are some important basic flutter concepts you really need to understand before getting our hands dirty with flutter block concepts.

So we will begin with these First, in order to make sure everything goes nicely into place.

Let me start by highlighting the proverb everyone interested in Flutter heard at least once.

Everything in Flutter is a widget, right?

So smash that like button if you knew that.

Well, since in this video we'll be discussing flutter block concepts.

You must know that every concept is actually a widget.

But what exactly is a widget?

Widgets in Flutter are classes classes which have their own attributes and methods.

These classes can be instantiated as objects by defining their required parameters and methods into their constructor.

As an example, let's take a look at the floating action button widget.

As you can see, there are all sorts of parameters passed through the constructor of this widget the foreground color, the background color, the onpress function, and many, many more.

If you tap control and click on the widget vs code will open a tab which contains the floating action button widget which, as I told you, is actually a class with all kinds of attributes from primitive types like booleans, strings, doubles, integers to functions and even other widgets.

You may have also noticed that some widgets have a child parameter.

Well, what could be that for?

Well, since everything in Flutter is a widget, there must be a way for the framework to keep them organized.

Right.

And what's a better way to nest and organize stuff than using a tree?

Observe this picture.

Take a look at the top of the tree.

That's the root node.

Below that, you can see two other widgets emerging from it.

These are the children.

The above widget is also called the parent widget or the ancestor, and that's how the tree evolves for every widget until there are no more children inside the tree.

Now that we know what a widget is and how Flutter organizes them, let's start with the first flutter block concept, which is block provider.

Now a quick note so that you won't get confused.

Remember that in the previous video I talked about qubits and blocks and the difference between them.

Well, there won't be a block provider and a separate qubit provider.

In this case, qubit is a smaller part of a block, so it works the same as a block with block provider.

Hence every flutter block feature I will discuss today will apply to both qubits and blocks.

So have you ever asked yourself after you build your first block, how will you access it in your UI?

How will you access it inside flutter in the first place?

You might say that you will create a block object for every widget where it may be useful if that's what you're thinking.

Well, it's actually pretty wrong.

I mean extremely wrong.

What if you have an internet block which manages the current state of the app in different network scenarios?

Think about how many widgets should have a working instance of the internet block available.

So creating an instance for every widget that needs it is a terrible idea.

Instead, what you must use is the block provider widget.

Block provider is a flutter widget which creates and provides a block to all of its children.

This is also known as a dependency injection widget so that a single instance of a block can be provided to multiple widgets within a subtree.

In other words, the entire subtree will benefit from a single instance of a block injected into it.

Hence the subtree will be dependent on the block we're providing.

So, for example, if we instantiate a block provider with a block A here at the top, it will be available

through the entire subtree below it.

If we instantiate it here in the middle part of the tree, the block instance will be available only to the nodes below and to none of the ones above.

Notice how block provider creates the single instance of the block using a function which takes the build context as an argument and returns.

The only one instance of block a build context is like its name is implying the context in which a specific widget is built.

So we'll have the current widget context and will be instantiating the block A to all of its widgets.

In the subtree, but how can we access it in the subtree?

The single instance of Block A can be accessed by calling block provider dot of block context or even simpler by calling context dot block of block k.

An optimization technique block provider offers is that by default it will create the block lazily, meaning that the create function will get executed when the block is looked up for via block provider of block a context so that there won't be a ton of created blocks when they're not needed inside the app.

To override this behavior and force the block provider to create function to be run immediately as soon as possible.

Lazy can be set to false inside the widget's parameters.

Take in mind that even though block provider is responsible for creating the block, it will also automatically close the block so that there won't be any stream leaks all over the place.

Okay, so this is what happens locally when you have only a screen in your app.

But what is it going to happen where there will be a new portion of the tree available to the existing one?

This scenario can happen when we push a new screen into the navigation routing feature of Flutter.

Is the block a going to be available there too?

The answer is no, and that's because there is another context created on the new route, a context which the block provider doesn't know yet.

So in order to pass the only instance of the block to the new route, we will use block provider that value in which we will pass the block to the value parameter, thus making the instance of it propagate through the next widgets.

Note that in this case the block provider that value won't close the block automatically, since we

still want the block to be accessible into the ancestor tree above.

Since we initially instantiated the block with block provider, it will, as I said, handle the closing part automatically.

This was a brief introduction on how block can be accessed inside Flutter, but we will have a separate video in which I'll talk more in depth about block access across every possible scenario.

Now, as an example, to make sure you understand the block provider concept by actually practicing it, we will start implementing the default counter app from Flutter using block.

So let's start by creating a flutter project.

You can name it whatever you want and then add the flutter, underscore block dependencies into the pubspec.yaml file.

Note that I'm also using the block vs code extension, so I recommend you to download it because believe me, it will make your life easier along the way.

Since the counter feature of the app we want to implement is really simple.

We will start by creating a new qubit called counter qubit.

From here I will start by coding an implementation that will be more extensive and self explanatory instead of a quicker and abstract one.

I'm doing this so that you will understand all the concepts really, really well.

So in the folder created by qubit, we'll open the counter state dot dart file and modify it so that it only includes the counter state class.

This class will be the blueprint for all the possible states which will be emitted by the qubit.

Since this class must have the counter value, let's write it down and also create a constructor for it.

The application must have a valid counter value at all times, so every time we'll have a new state emerging from the qubit, the counter value must be a required attribute, hence annotating it with the required keyword.

Now inside the counter qubit file.

Firstly, we need to set the initial state of the counter qubit.

Can you think about what the initial state is?

Well, it is definitely going to be a state, so we'll select the counter state class then.

As you can see, we'll receive an error inside vs code telling us that the counter value parameter is required, so we need to pass it inside the constructor.

The initial state of the counter app will have the zero value, so we'll set the counter value to zero.

Then all we have to do is implement the increment and decrement functions which will emit new counter states.

So the increment function will emit a new counter state with a new counter value passed into its constructor,

a counter value which is going to be equal to the current counter value plus one.

You can access the current state of a qubit by using the.

A state keyword.

This will return an instance of the current state of our counter and by saying state that counter value will receive the counter value of our current counter state.

The same procedure applies to the decrement function.

The only difference being that instead of adding one will subtract one from the current counter value.

Okay, so we finally created the counter qubit successfully by using everything we learned in the previous tutorial.

Now it's time to link the qubit to the UI using the first concept we learned today.

The block provider.

As you can see in the Main.dart file, the top of the tree is actually the material app widget.

This is where the tree starts from in most of the flutter apps.

So from what we've learned today, in order to make our counter qubit available throughout the entire tree of our app, we must create a block provider with it.

At our material app level.

So all we have to do is select the material app and then click control plus period to open up this dialog and then select wrap with block provider.

I told you the block vs code extension will come in handy then into the create parameter.

All we need to do is to take the current context and return a counter qubit instance.

Make sure you also import the flutter underscore block up there into the import section.

Okay, so what we did now is we told Flutter to create a single and unique instance of the counter qubit and make it available to the subtree below material app widget onto our home page.

We will adapt the code a little bit by adding a row child to the column widget.

A row which will contain two children.

The two floating action buttons which are supposed to increment or decrement the counter.

Remember what I said when we started learning?

Block that for every interaction a user makes with the app, there should be a state emerging from the

app, letting the user know what's going on.

Having that in mind, we conclude that for every tap of either of these two buttons, there should be a new counter state, perhaps counter value shown on the screen.

To do that, we need to call the increment or decrement functions on the qubit instance.

But how do we access the instance of the qubit?

As I told you before, either by calling block provider of counter qubit context or with context, dot block of counter qubit.

Then all we have to do is to pass the increment decrement functions onto the onpress parameter on the floating action buttons, which tells us that every time we press one of these buttons, the qubit dot increment or decrement functions will be called adding or subtracting one to the current state.

The question now is how do we receive the new state inside the UI?

How do we rebuild the specific text widget which prints out the counter value?

It is now time to introduce the second important flutter block concept, which is block builder.

Block Builder is, as its name suggests, a widget which helps rebuilding the UI based on some block state changes.

This is the magic component which rebuilds the UI every single time a new state is emitted by either block or qubit.

If you watch my previous video on block concepts, you have to know that this block builder is actually you waiting for the boats your friend sent down the river.

You are the block builder waiting for the boat and it's number so that you can rebuild the UI by calling the number out loud.

Now you have to understand that rebuilding a large chunk of the UI inside your app may take a lot of time to compute.

So that's why a good practice would be to wrap the exact part of the UI you want to rebuild Inside Block Builder.

For example, if you have a text widget which updates from a sequence of emitted states and that text is inside a ton of columns, rows and other widgets, it is a huge mistake to rebuild all of them just to update the text widget.

Instead, what you should do is rebuild only the text widget by wrapping it inside block Builder Syntax wise Block Builder is a widget which requires a block or qubit and a builder function.

The builder function will potentially be called many times due to how Flutter Engine works behind the scenes and should be a pure function that returns a widget in response to a state.

So what is a pure function, you may ask?

Well, a function where the return value depends only.

On the functions, arguments and no others, thus making it pure.

So in this case our builder function should return a widget, which only depends on the context and state parameters and nothing more.

If the qubit or block is not provided, the block builder will automatically perform a lookup for its instance using block provider and the current build context.

Now for even more control of when should the block builder rebuild the UI?

There is another parameter which you can use called build when this should take in a function.

Having the previous state and the current state as parameters.

For example, if the current state contains a text value and that text value is greater than the previous state text value, you can return false so that the builder function won't trigger and won't rebuild the UI.

Okay, so it is now time to integrate block Builder with our counter app example.

We left the app open inside the Main.dart file.

The counter value of the app should be printed right inside this text widget right here.

Now all we have to do is to wrap this specific text widget into a block builder by clicking control plus period and selecting block builder.

Note that I could have wrapped the entire center widget with a block builder instead, but that would be wrong performance wise, since the builder function will rebuild the entire widget tree instead of only the specific part we're interested in, which is the text value.

Okay.

So now into the builder function for every new emitted counter state, the text widget will show a new value.

The value can be accessed by calling state dot counter value and then by converting it to a string.

Since we know that the value is actually an integer, it is finally time to test out our app.

A moment which we've been waiting for so long.

So let's run it together.

As you can see, when I'm pressing the plus button, the counter qubit calculates the new counter value emits a new counter state with the new counter value, and then UI rebuilds with that specific value.

To make this rebuilding process even more clear, take a look on how I can play with the emitted states.

I can check whether the state dot counter value is negative and print a borough negative text widget.

I can check if the counter value is even and print yay.

Or I can check if the counter value is equal to five and print number five into the UI.

I hope you can now understand the power of streams states and block.

It may seem like Block is a little overkill for a counter app, but in order to understand the concepts,

we need to try them on simple examples and then see the power of it and what amazing apps and features

you can build by using it.

I hope you understood the block builder concept because it's now time to move on to another flutter

block concept, which is Block Listener.

Block Listener has mainly the same structure as block builder, but it is really different in many ways.

You see, the block builder function can be called multiple times inside the flutter engine due to how

Flutter Engine works behind the scenes.

So, for example, navigating to another screen can't be done inside block Builder as a response to

a state change because this action can be actually done multiple times.

Imagine the builder function inside being called five times.

Would you want the new screen to be pushed inside the navigation stack five times?

No way.

Or what?

If you want to show a snack bar or a dialogue for a new arriving state?

The builder function may be called four times by the flutter engine rebuilding the UI accordingly,

but the snack bar will be shown four times, as well as the dialog box, which pretty much is not what

we want.

This is where Block listener comes into place.

As its name suggests, this is a flutter widget which listens to any state change as block builder does.

But instead of rebuilding a widget as block builder did with the builder function, it takes a simple

void function called listener, which is called only once per state, not including the initial state.

Again, for fine grained control, there is also a listener when parameter function you can pass to

tell the block listener when to call the listener function or not as built when parameter was for the

block builder widget to understand block listener easily.

What we want to do is the following.

Every time the user presses plus or minus buttons, there should be a snack bar showing in.

In the bottom part of the screen saying incremented or decremented, depending, of course, on which

button was pressed.

Can you guess how we're going to do that?

Let's start with the beginning.

So what happens when we press either of these two buttons?

The increment or decrement function from inside the qubit gets called.

What do these functions do?

Well, they emit a new counter state with a new increment or decremented counter value.

So this is mainly where Flutter understands what the press button does.

And this is where we should tell flutter UI to a state that we incremented or decremented the value.

So all we have to do is to add another boolean attribute to the counter state class calling.

It was incremented.

This was incremented.

Field will be true when we press the plus button and false.

When we press the minus button, we can go into the counter qubit class and modify the increment or decrement function accordingly.

Setting up the was incremented parameter as we discussed.

And now we can go back inside our block listener and based on our state was incremented attribute.

We will display a snack bar accordingly to whether the counter value of our counter state was incremented or decremented and voila.

You can see from the screen that our application works as we wanted.

So perhaps you're in our situation right now when you're updating the UI using block builder and also showing a snack bar using block listener, Isn't there an easier way to do it?

Of course there is.

It's called Block Consumer.

Block Consumer is a widget which combines both block listener and block builder into a single widget.

How convenient.

So instead of writing block listener on top of block Builder, as we did in our example, actually we can refactor the code by writing both of them inside the block consumer widget and then copy and paste the necessary content from block listener to the listener function and the content from block builder to the builder function.

And voila, the code is no more readable and easier to maintain and the application works perfect.

Another amazing flutter block concept which you'll use a lot is repository provider.

Repository provider is the same exact widget as a block provider, the only difference being that it provides a single repository class instance instead of a single block instance.

We will talk about repository more thoroughly in future tutorials when I'll introduce you to the block architecture in a couple of words.

A repository is a class which has the main functions which make flutter, communicate with the outer data layer, the internet, the APIs, the databases and so on and so forth.

Again, we'll cover these concepts in depth in future tutorials in more feature rich applications.

You can imagine that there will be maybe tens of blocks in qubits having their own functionality.

In this tutorial we talked about how we can provide only one block or qubit to the UI.

But what are we going to do when we have ten blocks to provide and ten blocks to listen to?

When we want to display ten different snack bars, we will stack them together one after another?

No, we will definitely use multi block provider, multi block listener and even multi repository provider.

So these are mainly the simple concept widgets we discussed before.

But instead of taking only one block or qubit or even repository, they can take as many as they possibly can.

Okay, so it is finally time to wrap things up.

If it's the first time you saw these concepts, this might be a pretty difficult tutorial and I recommend you to watch this video at least twice and then try to build this counter app by yourself so that you can understand the flutter block concepts by practicing instead of just watching the tutorial.

Having all of these in mind is time to move to another chapter In chapter number four, I'm going to show you why it is crucial to have a clean and structured architecture in mind while designing and coding a flutter block based application.

Together, we'll find out what presentation business logic and data layers mean and what benefits do they bring in the long run?

With that being said, I'm looking forward to seeing each and every one of you up next.

Stay tuned and let's get Wicked.

4_bloc-architecture

1. BLoC Architecture

Hello, everyone, and welcome back.

In this chapter, I'll introduce you to the block architecture.

So remember a couple of tutorials back when I was talking about blocks and qubits.

I mentioned that we'll come back later into checking out what's up with this data layer.

We took out the equation back then.

Well, buckle up because the time has come.

First of all, what is an architecture?

What is it all about?

Why do we need an architecture in the first place?

Can't we just skip it?

Well, no, we can't skip it at all because as simple and as natural as it may be, if you think a little bit in depth, we as human species can't live without a predefined and stable architecture, which is mainly our skeleton.

Imagine having all kinds of different classes, methods, functions, variables all over the place.

This is definitely going to result in a total failure.

Now assimilate this to us humans.

How would our life be if we only had our organs?

Everyone had to choose where to place their heart, their lungs, their kidneys.

That's probably none of these people would have survived in this situation without an organized structure and stable skeleton.

Neither would have any of the apps built without a proper architecture.

It's as simple as that.

So think of the architecture as being the skeleton, the blueprint, the structure which keeps all your code organized, stable and easy to test and maintain.

Now, block architecture is simply an architecture which has block as its center of gravity.

So not only is block a design pattern and a state management library, but it is also an architectural pattern.

Let's come back to a familiar image.

So a couple of tutorials back.

What we know for now is that for every interaction an user makes with the app through each UI, there should be an event dispatch to the specialized block or qubit, which will process it and will eventually emit a state that is going to rebuild the UI in a way so that the user gets a feedback of what's going on with the app.

The big unknown variable in all this equation is how block processes the event and perhaps retrieves the necessary data to show to the user.

Let me put it this way for you.

Almost every app nowadays retrieves its data from the internet.

So in order to link our block based Flutter app with the outer data layer, we need to add the data layer into our equation so that, for example, whenever the block receives a fetching event from the UI, it will request some data from the internet, retrieve it as a response, parse it, and then return the data with a new state to the user.

We can split this into three separate main layers, which not surprisingly will depend one on another.

So the UI is mostly seen as a presentation layer, the block slash qubit as the business logic layer.

And the last but not the least, the apps data as simply the data layer.

We're going to start with the data layer which is farthest from the user interface and work our way up to the presentation layer.

Okay.

So as we discussed previously, the data layer has the responsibility of retrieving and also manipulating data from one or more sources, whether we're talking about network requests, databases or other asynchronous

sources to do this.

The data layer has been split up inside three important sublayers models, data providers and repositories.

I have set them up into this specific order because you will see that they will also be dependent on one another.

Okay, so you finish some flutter tutorials and you feel confident of what you have accomplished.

You want to build your own first app, Now you think of an idea, but then you realize that you don't know where to actually start programming the app.

What is actually the first thing you should code?

From my experience, I believe that the best way to start your application, of course, after you visually

and technically design it is by coding your models.

But what exactly is a modal?

A modal is as its name is implying, a blueprint to the data your application will work with.

Let's pretend your application is a brand new weather app, so the data it will be fetching from the internet is going to be mainly weather data, right?

Well, this data needs to be stored somehow inside your application.

How can you do that inside flutter simply by creating a class for the specific data.

So in this case, we'll have a weather class, right?

In this class, we'll need to set up specific attributes like temperature forecast, weather icons, wind speed, city name and so on and so forth.

But how do you know which attributes your class should have?

Well, you need to be sure that these attributes will be a little bit linked and not completely pay attention, a little bit linked to the data that will be received from the data source.

So if the data source for our app will be a weather API like AccuWeather, for example, chances are that the weather data response from them will be a.

Json string.

Telling you what the Json is is not a part of this tutorial, nor is the definition of an API, But in short, a Json is a collection of paired data like this one here and an API stands for application programming interface.

The API lets your product or app communicate with other products and apps without having to know how they were implemented in the first place.

So, for example, with a subscription we can access the AccuWeather API weather data without having to dive in into their code implementation.

We can just take the data we're interested in and that's all.

Therefore, a model is nothing more than a class which contains the data.

The application itself will be dependent on.

Note that it is highly recommended that your app models should be pretty independent of the source.

As I said, they should not be completely linked.

What do I mean by this?

The models from inside the app must be kind of generic and universal to multiple data sources.

So why shouldn't they be identical to the data provided by the weather API?

Well, think of what is going to happen if you have two different weather APIs in your app as a source

to your weather data, their APIs will set completely different jsons to your app.

Perhaps one of them has the temperature inside a temp name field and the other one in a temperature name field.

Then you will need to parse them separately to your universal models.

Remember that inside many apps there might be multiple data sources, but it's recommended to have only

one specific data model in which they will be parsed.

This means that your models should be independent enough to get weather data from many different weather

APIs.

So now, after you hopefully understood what a model is, it is time to move over to the next sub layer.

The data provider.

The data provider's responsibility is to provide raw data to its successor, the repository sub layer.

But let's concentrate on the data provider for a moment.

A data provider is actually an API for your own application.

This means that it is going to be a class which will contain different methods and these methods will serve as a direct communication way with the data sources.

This is where all the magic happens.

This is where Flutter asks for the required data straight from the internet.

All your network requests like Httpget http post put delete will go inside.

Take in mind that the return type of these functions won't be of the type of model you created earlier, but rather of the type of raw data you received from the data source, which for example may be a type of Json string.

The component in which we'll have our model class is instantiated as objects is the repository.

The repository is mainly a wrapper around one or more data providers.

It is safe to say that it's the part of the data layer block communicates with, similar to the other data layer sub parts repositories are also classes classes which contain dependencies of their respective data providers.

So for example, the weather repository will have a dependency on the weather provider, with the help of which we'll call the get row weather method and retrieve the weather row data.

The repository is where the model object will be instantiated with the row data from the data provider or row data which will be passed to model data with from Json methods.

The repository is also a perfect place for fine tuning the data before giving it as a response to the

block.

Here you can filter it, sort it and do all kinds of last moment changes before it will be sent to the business logic layer.

The business logic layer is where most of the blocks and qubits will be created inside your Flutter app.

Its main responsibility is to respond to the user input from the presentation layer with new emitted states.

This layer is meant to be the mediator between the beautiful, bright and colorful side of the UI and the dark, dangerous and unstable side of the data layer.

Why is one of them beautiful and shiny and the water one dark and unstable?

You might ask.

Well, you are supposedly designing and programming your app to the best standards so that the UI and the back end looks and most importantly works flawlessly.

This is the bright side.

But what about the Internet?

What about the APIs you're getting data from?

Do you have any guarantee that they will be as amazing and well programmed as your APIs?

Well, if the APIs are not designed and programmed by yourself, definitely no errors may arise.

Bugs, crashes, server issues and so on and so forth.

Being the mediator between the bright and the dark side, the business logic layer is the last layer that can intercept and catch any errors from within the data layer and protect the application from crashing.

The last moment.

Now, since this layer is closely related to the data layer, especially the repository sub layer,

it can depend on one or more repositories to retrieve the data needed to build up the application state.

Now see how every part of the puzzle starts to get into the right place now.

Truly amazing.

Another important fact you need to know and understand is that blocks can communicate one with each other.

Qubits can do this too.

This is really important.

So let's say we have our previous weather block and we also have an Internet block which emits states

based on whether there is a stable Internet connection or not.
Supposedly, your Internet connection dies when you want to know the weather from your location Inside the weather block you can depend on the Internet block and subscribe to its stream of emitted states, then react on every Internet state emitted by the Internet.
Block So in this case, the Internet block would have emitted a no Internet state down the stream.
The weather block will listen to the stream and will eventually receive the no Internet state.
The weather block can also emit a state in response to this Internet state, letting the user know that there is no Internet connection.
Don't forget that the subscription to the block needs to be closed manually by overriding the close method.
We don't want any stream leaks inside our app.
We have arrived at our final layer of the block architecture, the presentation layer.
This layer sums up everything related to the user interface, the widgets, the user inputs, the life cycle events, animations and so on and so forth.
And also its responsibility in our case is to figure out how to render itself based on one or more block states.
Most application flows will start with perhaps an app start event which triggers the application to fetch some data from the data layer.
For example, when the first screen of the app will be created inside its constructor, there will be an weather block which adds the app started event so that some location based weather will be displayed on the screen right away.
Here's a quick tip on how you can structure your project and folders in order to obey this architecture.
So the rule is that for every layer and even sub layer I introduced today, you need to have a separate folder inside the lib folder.
So in a series complete application you may have three folders data, business logic and presentation.
Inside the data layer, you might have multiple sub folders like models, data providers and repositories
Inside of the business logic folder.
You might have blocks and qubits of folders.
And finally, inside the presentation layer you might have multiple sub folders like widgets, pages, animations and so on and so forth.
You can name these folders, whatever you want as long as you stick to the architecture.
Okay, let's proceed to the entire block architecture workflow for a basic weather application.

We currently have only one screen and a single text input inside.

Let's say, for example, that the user wants to know the current weather for Chicago.

He types in Chicago and then hits the get weather button.

Currently, everything is happening inside the presentation layer.

Now when the user presses the get weather button, a get weather event was dispatched to the weather block.

Weather block knows that, first of all, it should emit a loading state to the user to let him know that it is working on his request by displaying a loading animation on the screen.

Then inside the weather block, there is a dependency on the weather provider which can be accessed and with the help of which the get weather for location will be called asynchronously.

We call it asynchronously because we don't know when the data is going to come back.

So we'll have to wait for it.

Now, inside the weather repository, there is a dependency for the weather API data provider, with the help of which we'll call the get raw weather for location of Chicago.

Here is where the connection with the outer dark parts of the internet occurs.

Inside of this get raw weather for location function, there will be an Http get request towards another API like AccuWeather, which will receive the request and then send back a Json string full of requested data back to our weather API inside the data provider layer if everything is okay, the raw weather data, which is actually a Json string, will be sent back to the weather repository where it will be parsed into a weather model.

Then finally the weather model will get passed back into the block.

The block is really happy that everything went smooth and emits a new state called Weather Loaded in which it will send the weather model object as an attribute towards the presentation layer.

The presentation layer, with the help of block builder will exit the loading state since a new.

State has been received and rebuilt.

The with the new weather data marking the end of this entire workflow.

I hope you have understood how important it is to have a well-designed architecture in place ready to organize code and scale your application over time.

Unfortunately, having a strong architecture doesn't mean that your application will be shockproof.

Bugs and errors may arise at the slightest scale refactoring, so it's important to have a strong testing method in order to make sure our app is as stable as possible.

That is what we're going to cover up in the next chapter of this course.

We're going to talk about block testing.

With that being said, I'm looking forward to seeing each and every one of you Up next.

Stay tuned and let's get Wicked.

5_bloc-testing

1. BLoC Testing

Hello.

What is going on, everyone?

Welcome back to this amazing course.

In this chapter, we'll take an in-depth look at how we can test a block build application.

Why is everyone so afraid of testing?

What are the pros and cons and why is it so important to test every feature of your app?

We'll discuss it all today.

So why do developers hate testing so much?

Well, the answer I'm always receiving is so simple.

We hate testing because the application works fine without it.

It seems like a reasonable answer, right?

And indeed it is.

An application can work perfectly fine without testing, but that doesn't mean we should just consider it so badly.

Testing is, in so many cases, a lifesaver.

It's the hero nobody talks about on the news.

So if you hate testing that much, think of it as a cute little dog, which will be loyal to you and your application all the time.

Think of it as a layer of security, a layer which will bark whenever something breaks inside your app.

Now, let me ask you something.

Have you ever worked on a big project in which you wanted to add a new feature and you realized that after two or even three days of work, despite the fact that the feature works perfectly fine, it breaks other features in a completely different place inside your app.

Or maybe have you thought if you ever want to delete a piece of code from within your app, will this action be destructive or non-destructive to the app itself?

How would you check if everything is okay afterwards?

Most of you will say that they will run the app and check every feature to see if everything works well,

but I'm sure that some of you won't even physically test it and live with the idea that nothing will break a conclusion which is fundamentally wrong.

Code is code and code will break.

Keep that in mind.

So what if I tell you that there is something which will run a full checkup of every feature you wrote inside your app so that whenever you refactor your code you will be stress free without having to worry about if it's going to crash or not.

You may have already guessed that I'm talking about our body testing.

Now.

The difference between the pros and cons becomes rather a problem of deciding whether it's worth choosing testing for its advantages over its disadvantages.

So obviously, we know that the disadvantages of writing tests are that in the first place we need to learn and understand the structure of them.

And in the second place that it will obviously take more time to implement and write all the necessary code.

But is it worth it?

I'll let you answer this question in the following minutes after we understand the structure and practice testing on our beloved counter app.

So testing.

How could we implement a test for a feature?

Think of it logically.

How would you test if something is working or not?

In big words, a test is defined by how you will programmatically tell Flutter to double check if the output given by a feature is equal to the expected response you planned on receiving and no other.

Now let's switch to the coding part where I'll show you how testing works and afterwards how to write tests inside your app.

So in order to follow up, you can clone my blog concepts counter project from GitHub because we'll start coding from there.

The link to that is in the description.

The first thing you should do after you open up the project inside vs code is to go and open up the pubspec.yaml file.

We already have the flutter underscore blog dependency inside, but for testing purposes we also need to add both the test and block underlying test dependencies to the record.

Now, inside our project we need to create a separate test folder in which we'll have all our test files.

A little hint I can give you is that every feature of your application needs to be tested in the first place.

So the test inside the test folder should be kind of symmetrical to the features from within your app.

So all we have to do is to create the folder symmetrically to the lib folder inside the test folder and add the underscore test to the name of the file.

In this case, we'll have a counter underscore qubit underscore test dot dart file in which we'll test out the functionality of our counter feature.

Remember from a couple of tutorials back that all the counter feature does is to increment or decrement a counter value based on which button is pressed by the user.

So inside the test file, we need to start by creating a main function.

Inside this main function, we'll go right ahead and create a group with the name of counter qubit.

A group is actually a way of organizing multiple tests for a feature.

So for example, inside our counter qubit group, we'll write all the necessary.

For the counter feature in a group.

You can also share a common setup and teardown function across all the available tests.

But what is the purpose of these functions?

Inside a setup function you can instantiate, for example, the objects our tests will be working with.

In our case, we will instantiate the counter block so that we can access it later on in our tests.

So setup is mainly, as its name is implying, a function which will be called to create an initialize.

The necessary data tests will work with.

On the other hand, the teardown function is a function that will get called after each test is run and if is called within a group, it will apply, of course only to the tests in that group.

Inside of this, perhaps we can close the created qubit.

Now that time has finally come for us to write our first test, which is going to be checking if the initial state of our counter qubit is equal to the counter state with a counter value of zero.

We're going to do this by creating a test function and give it a description which should denote the purpose of it.

In this case, it's going to be the initial state is counter state with a counter value of zero.

So how do we check this?

Well, I told you earlier that the purpose of any test is to double check that the output of the feature is actually equal to the expected output and nothing else to do this.

All we need is the accept function, which will take two main important arguments the actual value returned by our initial state and the expected value which we are expecting to be received.

So our initial state returned when the qubit is created, will be counter qubit dot state and the expected value should be a counter state which has the counter value equal to zero.

We can run this test by typing the pub run test command into the console or rather by clicking the run button next to the group.

So if we run this test, we will surprisingly receive a complete failure with this message in which we expected an instance of counter state and we actually receive an instance of counter state here, you can actually start to understand why testing is such an amazing tool.

It has already signaled us a problem with our app.

Remember that the application worked perfectly when we manually tested it.

How come that this test fails then?

Well, since it tells us that both the expected and actual output are an instance of counter state, and we know that both should have a zero value inside, that means that the instances are still different somehow, and that's due to the fact that inside Dart language two instances of the same exact class aren't equal, even though they are basically identical.

This is happening because these two instances are stored in a different part of the memory, and Dart compares their location in memory instead of their values, hence making them not equal.

You can override this behavior really simple by using a really popular library.

You may have already heard about Equatable in the back end.

Equatable is just a simple tool which overrides the equal operator and the hashCode for every class that extends it, hence tricking Dart into comparing the instances by value rather than by where they're placed in the memory.

So if you want the functionality of comparing two instances like we do in our test, equatable is the way to go.

Okay, so all we need to do inside our app right now is to add the Equatable dependency into the Pubspec.yaml file and then extend the counter state class with it.

Note that now we'll get a warning telling us that we need to override the props of the Equatable class.

The props are just a way of indicating equatable, which are the fields in our class that we want to

be compared while applying the equal operator.

So we're going to pass both the counter value and the was incremented attributes inside the props list.

And now whenever we will compare two counter states, Dart will compare them attributes by attributes in order to see whether they're equal or not.

So right now, if we go back to our test file and run the test, the test should finally pass since our expected and actual counter state have the same counter value, which is zero.

Now it is time for us to test out the functionality of the increment and.

Decrement functions from inside our counter feature because these are the most important right?

For this, we'll use the block test function from inside the block, underscore test dependency.

We'll use this because we need to test the output as a response to the increment or decrement functions.

We'll start with the test for the increment functionality.

So firstly we need to describe it properly.

The qubits should emit a counter state with a counter value of one and the was incremented value of true when qubit dot increment function is called.

To do this, we'll use the three parameters of build Act and expect from inside block test function.

The build parameter is a function that will return the current instance of the counter qubit in order to make it available to the testing process.

The act parameter is a function that will take the qubit and will return the action applied on it, which in our case is the increment function.

The expect parameter is an iterable list which will verify if the order of the states and the actual emitted states correspond with the emitted ones and no other.

Since the counter qubit emits only a single state.

In this example, we will place it inside the list accordingly.

The same procedure applies also to the decrement function, the only difference being that will act by calling the decrement function and that will expect a counter state with a counter value of minus one instead of one.

Since we subtract one from the initial counter value.

Let's run all the tests now and as you can see, everything passed.

Now we can know for sure that the counter qubit works as it should.

Think about this scenario, for example.

Now.

So let's say you moved away from this project and another guy comes into place to continue the development.

He doesn't know Dart that well and he thinks that the Equatable package is not needed and that the application

can work just fine without it.

So he refactors the code runs the app and indeed the app is still working perfectly.

But underneath, if we run the counter qubit test, we can see that if there would be a part inside

our app comparing two identical states, the output would have been completely different.

This is exactly why app testing should not be skipped.

It may seem like the refactoring works fine, but in reality it isn't.

So I guess now you understood why it's worth spending the extra time writing tests for your application.

In the next chapter, we'll dive into more advanced flutter block topics like block access and routing.

There are multiple ways of navigating between different screens in Flutter.

Therefore, there are also multiple ways of providing and accessing blocks or qubits between those pages.

This chapter is going to be the longest chapter from this course, so you might want to clear your mind

and grab some popcorn before.

With that being said, I'm looking forward to seeing each and every one of you in the next chapter.

Stay tuned and let's get Wicked.

6_bloc-access-and-navigation

1. BLoC Access & Navigation (Local, Route & Global Access)

Hey, what is going on, everyone?

Welcome back to chapter number six.

In this chapter, I will introduce you to block access more specifically in how you can access a block instance across various scenarios with block provider and block provider that value.

It is going to be a long but highly important tutorial, so make sure you buckle up and grab some popcorn.

At first.

Remember that in my third tutorial of this series regarding block concepts, I showed you how to provide a single block instance the counter qubit to the widget tree of a single page by using block provider.

Well in that video we only provide it locally to a single screen.

But what if we want to provide it to another screen, to multiple screens, or even globally to all of our app screens?

In order to follow up, you will also need to clone my repository from GitHub and open the third block concepts folder inside vs code so that we can continue right where we previously left.

Okay, so let's dive into our first scenario which will have actually discussed and programmed before locally providing a single instance of the counter qubit to our default home screen.

If you remember, if we wrap a widget with a block provider, a single instance of that specific block will be provided to all of its direct children.

We can provide it this way across the widget tree thanks to the build context, which is a way flutter knows which is actually the parent and which are the children across the widget tree.

Since most apps know this benefit from more than one screen, Flutter also comes with a navigation stack which organizes the routes and their specific screens inside the app.

So every time the app displays a new screen, we should know that Flutter pushes a new route with the new screen inside the navigation stack.

Every time we click the back button on our phone, we should know that flutter pops the current screen out of the stack.

There are many other actions besides pushing and popping which can be executed on Flutter's navigation stack, but these are beyond the purpose of this video.

You must know that every application must have a default home route, a route which is automatically pushed by default when the app is started.

In our case, inside the navigation stack, Flutter only pushes the default home route, which is the home screen containing the counter feature.

Coming back to our block access discussion, you may wonder.

Wicked.

You showed us that you can provide a single instance of the counter qubit to our home screen.

Why do you call it local access?

Well, it's called local access because as the name is implying, the qubit will only be available locally inside the widgets of that screen.

So you're saying that if we push another screen inside Flutter's navigation stack, the counter qubit won't be available there too?

Indeed, that's absolutely correct.

If we push a new screen into the navigation stack, Flutter will add its widgets to the already available widget tree.

But since these widgets are part of a new screen from a new route, the build context doesn't know how to indicate block provider where to provide a single instance of the counter qubit.

Up next.

So here comes the major breakup with local access.

Local access means providing a single instance of a block to only one screen, thus to the entire widget tree, which is part of that screen.

Let's test this theory for now inside our code.

So we know that currently locally, providing our counter qubit to our default home screen works perfectly.

Firstly, let's do some refactoring with the structure of our folders and code.

This way we'll also be testing the block architecture layers.

We learned a couple of tutorials back, so in order to stick to the architecture, we must highlight the business logic layer by creating a business logic folder, which I'll simply call it logic, and in which we'll move the qubit folder with our counter qubit.

Then, as we know, the UI should be part of the presentation layer.

So I'll create a presentation folder along with the screen subfolder.

Inside this folder, we'll place all of our different screens from inside this tutorial.

Inside the Main.dart file.

You can notice that as I told you previously, Flutter has a separate field inside Material Lab, which is named home, denoting that this is the screen which is going to be pushed up first onto the navigation stack.

In our case is the My Home page on which we'll press F2 and rename it to home screen.

We'll move the entire implementation inside a separate home, underscore screen dot dart file inside the screens folder.

I have also modified the home screen class a little bit so that it will take a title and a color as parameters in order to make it stand out compared to the next screens we'll create.

We'll also need to add a material button inside our home screen, which will push a new screen on the navigation stack.

When we press it, we'll leave the Onpress function empty for now to demonstrate that the counter block won't be accessible to a new push screen.

We would obviously need to create a new screen called second underscore screen Dot Dart In order to speed up the process.

I'll just duplicate some of our existing home screen code.

Inside our second screen class.

This is not really recommended in programming since we shouldn't have duplicate code, but for testing purposes, it's okay.

Now we need to go back to our home screen because we need to write the code to push the second screen

onto the navigation stack.

We'll do this by using navigator dot of context push and material page route to easily differentiate between the home screen and the second screen.

For the home screen, I'll use a blue accent color and for the second screen a red accent.

One.

Finally we can run the project and as you can see, everything is running fine.

The counter block is provided locally inside the home screen page.

All the functionality is working as it should, but if we press the go to second screen button, you will see that the application will crash with an error to be expected block provider that of called with a context that does not contain a bit of type counter qubit.

As you can see inside the second screen we're asking the block provider for the counter qubit instance.

But since the new build context doesn't know either where the instance is or how to find it, it throws

an error making the application crash.

You may be asking yourselves now, well, how will we provide the qubit instance to a new screen?

Then the answer to that question will be by using root access.

Until now, we only provided the qubit to a single screen, providing an instance of a block or qubit to only one screen is called local access.

Providing an instance of a block or qubit to multiple screens is called root access.

We're calling this root access because we will need to profit off the routing navigation feature of Flutter in order to provide the instance to the multiple screens.

You don't need to think of local and root access as two complete different scenarios because you will see that in the following part of the tutorial.

Some parts of the root access are linked to the local access.

Okay, so as I told you a couple of moments ago, root access is related to the routing navigation feature of Flutter because it's so strongly related.

I need to make a short introduction on how Flutter manages its routing feature.

Usually in a routing feature, you will need a pair of data containing both the routing name or tag and the page or screen you want to be displayed when you enter that root name.

Think of this as the way web pages work when you enter 500 pixels.

Dot com, which is a popular photography website.

If you want to navigate to the most popular photos, you can append slash popular to the website.

The slash popular is the root name and the popular page is what it gets loaded up.

After having this in mind, you need to know that in Flutter there are three major routing options available anonymous routing named routing and generated routing.

Anonymous routing is, as its name is implying, routing without a route name.

Anonymous routing is what we've been doing so far in our counter app.

So by default, we provided only the name of the home screen class to the home parameter inside the material app.

Also, when we navigate it from the home screen to the second screen, we used the material page route without having to provide a route name.

On the other hand, name routing is routing with a provided route name.

This is the routing method I explained a couple of moments before when I mentioned the example with the 500 pixels that come website, all the routes you want to provide have to be set inside the routes parameter of the material app widget.

Here's a quick example on how they will look like If you're building a small to medium sized application.

This named routing is recommended as opposed to the anonymous routing which might getrillioneally cluttered

and unorganized.

However, when you're working on a large project is recommended to use the generated routing.

Generated routing has almost the same structure as the named routing.

The only difference being is that you need to have a separate function in which you will have to set

up all our routes.

This is a more professional approach since it will declutter the material app of the routes parameter,

a parameter which might get pretty cluttered while working on a huge application.

You will see from the code that it's actually pretty simple and self-explanatory.

Since there are three major routing options, we'll tackle up each one of them to see how easy it is

to provide an instance of a block or qubit across each page.

In every of the above scenarios.

We'll start up with the anonymous routing since it's the one we're really close to right now.

Providing a block or a qubit instance across different anonymous routes is a key functionality, so

make sure you pay a little bit more attention to this topic.

Currently, we provided a counter qubit locally to our home screen, therefore to all of its children

widgets.

We know that when we press the go to second screen button there will be a new anonymous route pushed

on the navigation stack, loading up our second screen.

Unfortunately, due to the fact that the second screen has a different build context, the block provider

isn't able to.

Continue providing the counter qubit instance to the second screen, thus making the application crash.

So there is a working instance of the counter qubit available in the entire home screen widget tree.

How can we provide this working instance up next to the entire widget tree of the second screen?

That's the big question.

The answer is by using block provider that value widget block provider that value takes an existing

instance of a block or qubit and provides it further to its child widget.

In our case, the second screen.

Let's switch to the coding part so that you'll understand the concept better.

So we're inside the home screen class, right where the second screen anonymous root gets pushed on

the navigation stack.

This is done by using material page route.

The second screen is being returned by the builder function.

So right before that, we need to wrap the second screen widget with the block provider that value widget

so that we will provide a counter block instance.

Further on.

Now, please pay the most attention you possibly can because I'll be telling you some important information

you'll definitely forget if you aren't paying much attention.

There are two important steps you really need to pay attention to.

Remember that I told you that the block provider that value widget takes in an existing instance of

a block or qubit and provides it further down the next screen's widget tree.

The existing instance will be passed right inside the value parameter from the block provider that value

widget.

This is where most people, including me, make a huge mistake at first.

They write block provider dot value and they provide the value of counter qubit.

If you save this project now and run it, you will see that everything works fine.

The error is gone.

All the functionality seems to be working smoothly now while being on the home screen.

Let's increment the counter to five and then go to the second screen.

You may notice that something isn't quite right.

The counter is zero, but if we go back, the counter is still five.

Why is this happening?

Take a look at what we actually provided to the value parameter.

A new instance of counter qubit.

What should have we provided instead?

An existing instance of the counter qubit?

So what happens is that every time we push the second screen, we wrongly provide a new counter qubit

instance.

This is really bad because not only the instance won't close automatically since it was not created

by a block provider, but it will also have a completely different state than the existing instance

from the home screen.

And it will defeat the purpose of having one unique instance of the counter qubit around our app.
What we should do instead is provide the existing instance of the counter qubit to the second screen.
If you remember back from the Block Concepts video, we can access an existing instance of a qubit or block by calling either block provider of qubit, context or context block qubit.
This was one of the two problems that may encounter if you're not careful when you're providing blocks or qubits to new pages.
Let's say that this one can be avoided, but the next one is simply too easy to mess up.
You may think that if you save and run the project now, everything will work.
That's what I also thought multiple times.
But surprise, you'll see the previous error popping up again, this time arising from the builder function inside the material page route instead of the second screen.
How could an flutter find the existing instance of the counter qubit in the context we know for sure that's available inside the context of the home screen.
But is the context from which we're searching for the counter qubit the appropriate context?
No, it's not.
If you look closer, the context in which we're searching for the counter qubit is the new context.
The second screen is being created with.
So instead of searching inside the home screen context, we are wrongly searching inside the new context of the second screen.
That's because we have the same name for both contexts.
Imagine having to debug this.
It would have been a nightmare.
That's why I told you to pay as much attention to these two major issues that may arise.
In order to fix this, we have to rename the second screen context to new context or maybe replace it with underscore since we won't use it's value now.
After we save and run the application, we can see that everything works as it should.
We successfully provided the only instance of counter qubit to the second screen's anonymous root.
If you're developing a small application with a few screens, this is how you can provide an instance of a block or qubit to all of your anonymous screen roots one by one on multiple screens, though, you may need to take a look at named or generated root access.
And that's what we'll tackle up next.
But first, let's create a third screen for the next topic.

We'll duplicate some parts of the home screen again just to save some time.

Okay.

So named root access is basically referring to providing an existing instance of a block or qubit to the screens while navigating with name roots.

Since we're currently navigating with anonymous roots, we need to completely refactor the navigation part of our app in a big.

Application refactoring.

The navigation feature may take a huge amount of time.

So that's why it's important to stick to a navigating technique from the start.

Depending on how big your project is is going to be.

We will list all our routes inside the routes parameter of our material app class.

Any application should have a default home route which will be pushed in the first place when the application starts.

This route must be named Slash inside the routes list.

Since Slash is the universal sign for home in programming, let's get to the coding part since it would be easier to explain what's going on.

Let's start refactoring the entire code in order to integrate named routes.

Since we have already created the screens, we need to decide which one is going to be the home route and then associate the rest of them with a route name.

As I told you, the material app class comes with a routes parameter which takes in a map of pairs pairs which contain both the route name and the screen objects.

We want to assign them to.

Okay, so first and foremost, we need to set up our home route by routing its route name, which is slash and link it to the screen.

We want to launch when the route is pushed down the navigation stack.

Since the screen we want to push is the home screen.

We'll write down a function, taking the build context as a parameter and returning the home screen.

The same procedure applies for the rest of our screens.

Let's name Slash second, the round name, referring to the second screen and slash third the route referring to the third screen.

Of course, in order to easily differentiate between them, I will set them different colors and titles

so that the second screen has a red accent color and the third screen has a green accent color.

These are all our routes and lane screens will use inside our app.

Since we have a routes attribute with an already defined home route, we can delete the home parameter since we won't need it any longer.

Now for this named route access topic, we're interested in how we can navigate by using named routes, but most importantly, how we can provide a unique instance of a block or qubit across all of them.

You might notice that right now we don't have any block provider widget wrapping any of our screens.

Can you guess why?

Well, if we had, we would have to provide a new instance of our counter qubit for every screen.

Again, we definitely don't want this.

We want to provide a unique instance of our counter qubit across all of our screens.

So this time we'll create the instance manually.

Here at the top we will prepend the name of the variable with an underscore to make the field only available

inside this class.

Now that we have a unique instance of counter qubit, can we provide it with block provider across every screen?

Well, yes you could, but it's not really worth it since the only advantage of using block provider is that it automatically closes the counter qubit, but only when the counter qubit is created inside the block provider itself.

We created it at the top of our my app class.

So using block provider would be nonsense.

Then what should you use instead?

Take in mind what I have already told you.

When we want to provide an existing instance of a block or qubit, we will use block provider that value widget when we want to create and provide a block or qubit in the same place.

We should use the standalone block provider widget.

Here.

We can't create and provide the counter qubit for every screen because that would lead us to having three different counter qubit instances.

I have told you multiple times that inside any app we should only have one unique working instance of a qubit or block.

So that's why we created only one here at the top and that was it.

Hence we need to wrap all of these screens with a block provider dot value widget and provide them with our existing single instance of counter qubit.

Remember that block provider dot value won't close the provided block automatically.

We will need to close it manually where we created it.

More specifically, where the My app class gets disposed.

My app is a stateless widget which doesn't have a dispose method, so we'll have to convert it to a stateful widget.

And then inside the dispose method we will need to call the close function on our counter qubit where almost done.

All we need to do now is to modify the buttons from where we pushed our name routes into the navigation

stack.

For that, firstly, we will need to visit the home screen from where we'll have two buttons one navigating to the second screen and the other one navigating to the third screen in order to push a named route down the navigation stack, we need to use the navigation that of context dot push named function and write the specific route name as an argument.

Now, if we run and save the project, we can see that everything is working perfectly as expected.

The navigation.

Feature is fully functional.

The counter feature works flawlessly and this is how you can provide a block or qubit to multiple screens of your apps by using named routing.

Now we can finally get to my favorite way to navigate inside Flutter, which is by using generated routes.

So why is generated routing my top choice when it comes to navigating in Flutter, you may ask?

Well, mainly for two main reasons.

One is that it mainly looks the same as the name routing, which means it borrows its advantages, thus being organized and properly structured.

The second thing is that everything related to the navigation can now be separated into a single file.

Okay, since we're going to change the navigation feature again inside our app, we'll need to refactor the navigation part again.

We'll begin by creating a separate router folder inside the presentation layer.

Since overall navigation is related to the UI.

Inside the folder, we'll create a file called app Underscore Router Dot Dart in which we'll set up all the logic for navigating to all of our screens.

The main idea here is that we want to create a function which takes in mainly the name of the route we want to generate and returns the generated screen accordingly.

So we'll create a function called on Generate Route, which returns a type of route, and this is going to accept a route settings parameter.

Inside this function, we'll create a switch statement which will contain our three route names.

The route names can be accessed from inside the route settings parameter by calling settings dot name.

So we'll have three main cases one for the home screen, one from the second screen, and finally one for the third screen.

For each case, we will return a material page route which returns our specific screen inside its builder function.

Since this is a switch statement, we also need to mention a default case which will be called whenever one of the route names is not equal to any of the mentioned cases.

Okay.

So again, we set up the navigation feature, but what we want now is to know how we can provide an existing instance of a block or qubit to each of our screens.

Creating a new instance for each screen is again not an option.

So instead we will create the unique qubit instance right at the top, as we did with the name routing.

Then, since we need to provide the existing instance of qubit we created to all of our screens, we need to wrap them inside a block provider that value widget.

Again.

Since we created the qubit manually, we'll also need to manually close it.

We'll do this by creating a dispose function in which we'll close the qubit.

We're close to finishing this scenario.

Our app router class is ready to manage the navigation feature of our app, but now we need to go inside the manger file and in the material app we can delete the routes parameter completely since we don't need it anymore.

Instead, we'll use the own generate route parameter which takes in our own generate route function from inside our app router class.

So first of all, we need to create an app router instance at the top and then pass the own generate route function of it to the own generate route parameter.

Pay attention because we need to pass the function as an argument and not as the results of it.

Finally, inside the dispose function we'll close the counter qubit by calling app router dot dispose.

I want you now to observe how organized everything is.

Every feature is placed in its own folder and is structured nicely and easy to read and understand.

If you run the project now, you can see that everything is working as expected.

Now think of this fact for a moment.

Inside your application, you'll definitely have multiple blocks and qubits.

Each one of them will have their own functionality and will work on a specific screen or part of your app.

Now you know how to provide our block instances locally and specifically to each of your multiple screens.

But what happens if you have 100 screens and you want to provide a block to all of your screens?

Will you have to provide it one by one?

Think a little bit about an authentication block.

You need to know the state of it around the entire app, not only inside the login or logout page.

Why?

Well, what if the authentication session expires?

You will want to kick the authenticated user out of whatever the screen he is browsing at that moment.

Right?

In this case, we'll need global access, but how can we provide a single instance of a qubit or block to all of our screens?

You wouldn't believe this, but if you wrap the material app inside the block provider or even inside the multi block provider, every block or qubit you provide there will be globally available inside every screen of your app.

Why is this possible is beyond the purpose of this video, but it will be addressed in detail in future tutorials.

Let's tackle up the coding part.

So right now we're manually providing an existing counter qubit to all of our screens.

One.

Buy one because we need it in all our screens.

But what we can do instead is provide it globally.

As we discussed, let's refactor the code for the last time in this tutorial.

We don't need this instance anymore here and none of the block provider that value widgets.

We can also delete the dispose function since we'll create the instance of the block inside the block provider.
Therefore the block provider will automatically close it.
The My app class does need to be a stateful widget anymore, so we'll convert it back to a stateless widget.
All we need to do now is to wrap the material app inside the block provider and create the only instance of the counter qubit to be provided globally to all of our screens.
As you can see, if you save the project and run it right now, everything will work as previously,
but this time having the counter qubit provided globally accessible from every screen inside the app.
Okay, I know it's been a long tutorial for you guys and I hope I didn't lose you on the way here.
There is some important information you need to have in your mind before you leave this tutorial.
I recommend you to watch it multiple times and practice as much as you can.
Here goes a list of what you need to briefly understand after you watch this tutorial.
So by default, you can navigate in many ways inside flutter by using anonymous routing named routing and generated routing.
Remember that the key here is to provide a unique instance of a block or qubit to one or multiple screens.
You shouldn't create multiple instances of the same block because that may cause unexpected behavior.
Take in mind that block provider creates and provides a new instance of a qubit or block.
On the other hand block provider that value takes an already created instance of a block or qubit and then provide it.
Furthermore, depending on the navigation way you chose, you can provide your block or qubit instances the following ways locally when you want to provide the instance to a single screen.
Specifically when you want to provide the instance specifically across one or multiple screens and globally.
When you want to provide the instance across all of your screens.
And remember to always double check in which context you are searching for an existing instance of the block or qubit, because that can also cause a lot of problems.
Wow.
We have finally arrived at the end of the sixth chapter.
I hope I didn't lose you along the way.
I admit it was a long and difficult one, so please feel free to rewatch it as many times as you want.
However, in the next chapter we'll relax a little bit and talk about how two or more qubits or blocks communicate one with other ones.

We'll definitely like this one.

With that being said, I'm looking forward to seeing each and every one of you in the next chapter.

Stay tuned and let's get Wicked.

7_bloc-to-bloc-communication

1. Bloc-to-bloc Communication

Hey, what is going on, everyone?

And welcome back to chapter number seven.

In this chapter, I will introduce you to block to block communication, especially how blocks or qubits communicate between each other.

In this tutorial, we'll also cover up some further block architecture tips and tricks and do a general recap of everything we learned so far so that you'll be able to practice and structure your code even better.

Also, if you want to follow up by practicing this tutorial, you need to clone my block from zero to Hero repository from GitHub and open up the sixth block access folder since that's where we left last time.

Okay.

So in order to start this lesson with a motivational purpose, you need to know that sooner or later inside your application you will have one or multiple blocks or qubits which may need to communicate with other blocks or qubits.

By communication I mean that 1st May do something in response to a state change of another.

Remember from my third and second tutorial on this series that both qubits and blocks emit a stream of states?

Well, because of that, other blocks can listen and receive those states and then act appropriately in response to them by perhaps adding an event or by calling a new function.

So the main question here is how can a block or qubit listen to another block or qubit?

To be fair, there are two possible scenarios you can tackle in order to achieve this.

Both have their own pros and cons and can be equally good, so the method you'll be choosing in the future will be strictly a matter of taste.

Today, as an example, we'll learn how to communicate between two qubits by adapting our existing app a little bit so that whenever the phone is connected to Wi-Fi, the counter will be incremented by one.

And whenever the phone is connected to mobile data, the counter will be decremented by one.

So we'll have an internet qubit and a counter qubit, both communicating one with another.

Let's start by discussing the first method on how two blocks or qubits can communicate one with another.

If you remember back in the tutorial number two, when I talked about streams, I told you that in order to listen to a stream of data, you will need to subscribe to it by using a stream subscription.

A stream subscription is the foundation object from which you can listen to a stream.

So what would stop us from using it as a method to listen to a block or qubit?

Well, nothing.

So currently we know that blocks or qubits start to emit states right after their unique instance is created.

Most of them will emit an initial state right after they were created.

So it's important to understand that the stream subscription object will use to listen to a block or qubit should be listening right from the start and actually before the first state is emitted.

How can we do that?

Well, by initializing the stream subscription inside the block or qubit constructor.

So in our example, we'll set up an internet stream subscription object inside the Counter Qubits constructor.

Since this is where we'll start listening to the internet blocks emitted states.

However, you need to understand that since you created a stream subscription manually, you'll also need to close it manually somewhere so that you won't listen to the stream forever.

You can do this inside the overridden close method from inside any block or qubit.

Okay, so we established the connection between the qubits, but what's the purpose of it?

The purpose of it lies in the listen function.

There you'll have a stream of internet states coming from the internet qubit.

So since we want the counter incremented by one every time the phone is connected to Wi-Fi, all we need to do is checking if an incoming state is the one corresponding to when the phone is connected to Wi-Fi and as a result will count the counter qubits increment function right away.

This is how you can use stream subscription in order to establish a block to block connection.

Let's dive ourselves into the coding part.

Okay, so after opening project number six Inside vs Code, we'll start by browsing the Pubspec.yaml file here.

We'll need to add the latest connectivity underscore plus dependency to our project.

This dependency will help us find out when the phone is connected to either wi fi mobile or to nothing.

Now that on Android, if the plugin reports a Wi-Fi or mobile connection running on your phone that

doesn't mean it has internet connection.

For now, we're only interested if the phone is connected to a network or not, and we don't care if that network sends or retrieves data from the internet.

Now let's start by creating our internet qubit and move it inside our qubit folder.

The first thing I always code after creating a block or qubit are its states.

Because first and foremost, you need to know the purpose of your block or qubit.

And also what are the possible cases it will return as a stream of data.

So by analyzing the documentation of connectivity plus package, we can deduce that the first possible states are connected to Wi-Fi, connected to mobile and disconnected.

Since there are two almost identical states, we'll create an enum for the type of connection we're using.

There are two types, so we'll have two members for the connection type enum, Wi-Fi and mobile.

Now here is an architecture tip number one predefined types, enums, constants, and generally code that.

Doesn't change at all during the life cycle of your app can be moved inside the constants folder.

We'll create one inside the lib folder.

We also need to create an enum start file.

Since every enum you'll create inside your app must lie down there for easier access.

Therefore the code will be more organized.

Okay.

Coming back to our internet states, we have decided to have an internet connected and internet disconnected state.

We'll focus on the internet connected for a moment.

We know that our phone can be connected to the Internet by using either Wi-Fi or mobile data.

Therefore, the type of connection needs to be specified inside the state.

This can be done by creating a final variable of the connection type.

We'll assign it as a required parameter inside the constructor.

So whenever the internet qubit will emit a new internet connected state, we'll have to mention the type of connection we're using, whether it's wifi or mobile.

By default, every block or qubit comes with an initial state.

This state is emitted when the block or qubit instance gets created.

We know that we'll instantiate the internet qubit when the application starts and since the plugin may take some time to detect whether the phone is connected to Wi-Fi or mobile or to nothing, we'll need to show a loading animation in order to let the user know that something is actually happening in the background.

So instead of showing an internet initial state, we'll show an internet loading state.

Now, if we switch to the Internet qubit class, we can notice that when the internet qubit constructor gets called the first emitted internet state down the stream will be internet loading.

Okay, so we finished coding our internet state successfully.

From this point, I like to continue by creating the specific qubit methods which will emit the states we created previously.

The emit internet connected function will take a connection type as a parameter and will emit an internet connected state with the same connection type so that we know whether it's the Wi-Fi or mobile data we're connected to.

Last but not least, we'll create an emit internet disconnected function that will emit the internet disconnected state accordingly.

So to recap, the purpose of this internet qubit is to emit different internet connection states based on what is returned by the connectivity plus plugin.

Therefore, somehow we need to react to do something in response to the connectivity plus plugin.

So we need to add a dependency of it into the internet qubit class.

We can do this by creating a connectivity variable and passing it as a required parameter Inside the constructor.

Let's take a look again at how the plugin returns, the type of connection of your phone.

As you can see, it can either return it a single time when we call the check connectivity function or multiple times through a stream that is internally managed by the plugin.

We're really interested in the latter scenario since we always want to know the state of our connection.

Hence we want to listen to every possible connection type the plugin sends down the stream.

How convenient.

So today not only will learn how a block can listen to another block, but also how can a block listen to a specific stream of a plugin?

Both have the same stream subscription approach, so it's amazing that we can tackle out both of them.

Okay, since we need our internet qubit to listen to the connectivity plus stream, first and foremost, we'll need to create a stream subscription so that we'll be able to subscribe to the specific stream.

Then we'll need to initialize this connectivity stream subscription inside the constructor by subscribing to the on connectivity change stream and then by listening to it.

So to recap, every time a new connection is noticed by the connectivity plus plugin, this on connectivity change stream will send the connectivity result down the stream.

Currently, our internet qubit is just listening to the connectivity results emitted by the plugin, but we also need to do something in response to them.

We need to emit new internet states accordingly to what the connectivity result is.

So every time a connectivity result is received down, the stream will verify if it's of type connectivity, Wi-Fi, mobile or none.

If it's of type connectivity, Wi-Fi, we will call the emit Internet connected function from inside the qubit.

In the mobile case, we'll change the connection type to mobile.

And finally, if the connectivity result is of type connectivity result dot none will emit the internet disconnected state.

Note that we don't need to emit the internet loading state since it will be automatically emitted when the internet qubit is created.

If you hover on the connectivity stream subscription variable, you can see that Dart is warning you because the created stream subscription hasn't been closed yet.

We don't want it to be listening forever, so we'll close the subscription right when the block is closed.

So let's have a little overview of what we did until now.

So we added the connectivity plus dependency in our project.

We created the internet qubit with its respective Internet states inside the Internet qubit.

We created the methods from which we can emit our created Internet state.

And then we learn how a qubit can communicate with a plugin by subscribing to its stream of data.

The stream contains the type of connection our.

Phone records during the life cycle of our app.

Depending on whether the phone is connected to Wi-Fi, mobile or to nothing, we called our qubit functions.

Accordingly.

Nice.

We're really making a ton of progress.

What's left for now is how our well known counter qubit can communicate with a newly created internet

qubit.

Remember that our goal is that whenever the phone is connected to Wi-Fi, the counter should increment, and whenever the phone is connected to mobile, the counter should decrement.

So let's switch our attention to the counter qubit since we need a counter to be incremented or decremented

as a result of a new internet state, the counter qubit will be dependent on the internet qubit.

So we'll need to have a final field of type internet qubit ready to be initialized inside the counter qubit constructor as a required field.

Then, in order for the counter qubit to communicate with the internet, qubit will proceed as we did previously, by creating a stream subscription which will subscribe to the internet qubit stream of states.

Remember that each block or qubit has a stream of states you can subscribe to.

Therefore, internet qubit has one two and will subscribe to it inside counter qubit so that whenever a new internet state is retrieved down the stream, we can do something in response.

In our case, we need to check whether the received internet state is a type of internet connected and the connection type is either Wi-Fi or mobile so that we can call either the increment or decrement functions respectively.

Again, we'll need to cancel the Internet stream subscription inside the close method of the qubit so that we won't listen to it forever.

And now here's architecture Tip number two.

You have to agree that the code looks a little bit messy now, so let's refactor it a little bit.

Inside the Internet qubit, you can see that all we do in the constructor can be extracted to a method.

We can call it monitor Internet connection.

Since this is what the function actually does, we can do the same with the counter qubit.

This time we'll call it monitor internet qubit since according to the states emitted by it will increment or decrement our counter value.

All we need to do now is to provide both our internet qubit and counter qubit globally to our pages.

But the reality is that the counter qubit is dependent on the internet, qubit and the internet qubit is also dependent on the connectivity plus plugin.

So what do we do?

In this case, What I usually do is instantiate the objects by following the order from the least dependent one to the most dependent one.

We know that our connectivity plugin doesn't depend on anything, so we'll create it.

Firstly, and now here's architecture Tip number three A good practice when you want to create a standalone

instance, which is an instance that doesn't depend on anything, is to create it at the top, inside

the main function and then inject it into the app.

In our case, within the My app class.

This way our dependent qubits blocks or repositories can use their specific methods like block provider

repository provider to create and also inject themselves into the rest of the app accordingly, while

also having access to the required dependencies of course, this varies on the way you chose to provide

your blocks.

As we learned in the previous tutorial, all you need to do is to adapt to your needs.

Now we need to modify the UI of the home screen a little bit just to double check that everything works.

We'll also place a block builder which will rebuild a text widget according to the current internet

state.

So whenever the internet state is internet connected, we'll display either Wi-Fi or mobile.

Also, when the Internet state is internet disconnected, will display a disconnected text widget on

the screen.

Finally, when the Internet state is internet loading, we'll use a circular progress indicator to let

the user know that some process is actually running in the background.

We'll also disable the plus and minus buttons since we won't use their functionality anymore.

I also did some refactoring to the code which only beautifies our app.

You don't need to do that since we only care about the functionality right now.

We can finally save and run the application.

And as you can see, when it finds out that the phone is connected to Wi-Fi, a new internet connected

state is emitted and the UI rebuilds accordingly.

You can also notice that the counter qubit successfully retrieved the new internet state too, since

it incremented the value by one.

And if we turn off the Wi-Fi and connect to mobile data, you will see the same behavior.

This time, though, the counter will decrement when no internet connection is available, though the

application displays it properly and the counter doesn't do anything.

Since it wasn't programmed to do anything as response to the state, we finally succeeded tackling up

this block to block communication scenario by using stream subscriptions.

I hope that from the above code you learned even more than that.

Since we also improved our block architecture structure even more.

Let's have a minute and talk a little bit about the pros and cons of this stream subscription way of communicating between blocks.

Well, for me, the advantage is that the code looks really understandable, readable and easy to structure and maintain.

This method will also help you practice your stream skills since.

Subscription is the foundation of subscribing and listening to the stream flow.

A disadvantage may be that if you have lots and lots of block to block communications, you may be overwhelmed

by the number of stream subscriptions you need to create, but most importantly, close.

Things can get pretty cluttered in a big application and huge memory leaks may appear as a result.

So if you're not really ready to manage the stream subscriptions, you'll need to take a look at the next method used to communicate between blocks and qubits.

Remember that in my third flutter block concept tutorial of the series, I specifically introduced you to a widget called Block Listener.

If you remember, this widget was meant to be used only when you want to do something in response to a block or qubit state change.

For example, we use block listener to show an increment or decrement Snackbar.

So what prevents us from using the block listener in order to do something in response to a new state?

Absolutely nothing.

But why would we do that?

What is the advantage of this over the Stream subscription method?

Well, the advantage is that you don't need to create a stream subscription anymore since the block or Qubit manages it internally from the start.

So on having many blocks communicating with each other.

You don't need to think of which subscription you should cancel or create.

Everything is being taken care of under the hood.

At the first glance, since the block listener is a widget and must be placed inside the widget tree, you may feel like doing something in response to a state change inside the UI introduces logic.

Inside the UI, therefore violates our architectural principle of isolating the logic from the UI.

This is wrong because calling a new qubit function or adding a new block event isn't related to the

logic of the app.

It's just a notifier on what the block or qubit should do.

It's just notifying the block what to do, not how to do it.

Hence, by using the block listener widget in order to do something in response to a state, change isn't related to logic.

Okay, let's get into the coding part of this method.

So definitely the most important advantage over stream subscription is the fact that we don't need to create or manage the stream subscriptions manually.

So let's refactor the code and delete the subscriptions from inside the counter.

Qubit.

We don't even need the internet block dependency anymore since we'll just listen to it by using block listener inside the widget tree.

So the counter qubit class will actually be having just its plain functionality in there.

But now inside the home screen page, we can add a block listener somewhere here.

Doesn't really matter where, as long as there is a context containing the block or qubit, you want to react to the incoming states.

So in our case, the context should contain both the internet qubit and counter qubit.

Unique instances we provided them globally, so they're available everywhere inside the app.

I like to wrap the top of the widget tree of the page with the block listener so that it looks more organized.

Remember that block listener is different to block builder block listener just listens to a stream of data while block builder listens and rebuilds the UI based on a stream of data so it's safe to place block listener at the top of the tree, since it won't rebuild anything.

So Inside the Block Listener We'll listen to the internet qubit and act accordingly by coding the behavior that we want.

If the Retrieved internet state is of type internet connected and the internet type is wi fi, we'll call the increment function of the counter qubit.

If it's of type mobile, we'll call the decrement one again.

We can access the counter qubit unique provided instance by either calling context, dot block or provider dot of.

Now, if we save and run the project, you will see that the application has exactly the same functionality as with the stream subscription method.

So what about the pros and cons of this block listener approach compared to the Stream subscription one Inside Block Listener Block Library takes care internally of all its stream subscriptions, so you don't need to spend extra time verifying where they were created or closed in the first place.

In the same time, adding block listeners inside your UI may also create some readability issues if you have a lot of them stacked one over another.

So at the end of the day, it's up to you.

What's the method you'll be approaching more or less.

Let me know in the comment section which one you actually prefer the most.

I hope you enjoyed this chapter.

In the next one I will get really in-depth on how build context works.

This is an important topic.

I noticed most of the tutorials fail to explain properly.

We'll understand together how Flutter builds the widget tree in the background and how build context helps towards organizing and structuring it.

Believe me, if you have any doubts on how build context works along with flutter and block, after watching this next chapter, you won't have anything more to worry about.

With that being said, I'm looking forward to seeing each and every one of you in the next chapter.

Stay tuned and let's get Wicked.

8_build-context-in-depth

1. Build Context In-Depth

Hey, what is going on, everyone?

And welcome back to chapter number eight.

In this chapter, I will dive even deeper into the concept of build context, talking about how it works inside Flutter and therefore how it is used by the block library to provide different blocks or qubits across the widget tree.

This is a really, really important tutorial, so make sure you don't skip it.

We will be going to tackle up different scenarios to see how build contexts can cause the unwanted block provider that of fails to find the context containing a specific block or qubit error.

So if you encounter this issue too often, it's a great idea to watch this tutorial in depth.

Remember from my third tutorial of this series regarding flutter block concepts?

When I told you that everything in Flutter is mainly a widget and that Flutter uses what's called a widget tree in order to organize and structure the widgets you want to set up and use inside your application.

But how is this widget tree structured at the end?

What links everything into place in a perfect harmony?

You guessed it, the build context.

So the build context is somehow a tool which helps handle the location of a widget inside the widget tree.

So you may ask yourself, Well, that means every widget is built within a build context, right?

Absolutely correct.

In fact, every widget has a build function, a function which gets called whenever the widget is being built in your application.

Inside that builder function, you will often see a build context instance passed that lets flutter know where the widget will sit inside the widget tree.

You must know that not all widgets have a user accessible context by default.

For example, a text widget will be built within a context, but that context will be anonymous by default.

On the other hand, when you're creating a block provider or whenever you're building a new screen inside the build function parameters, you will see that there is a build context instance that you can use

later on in order to access the context inside your page.

To better visualize how build context actually works.

We will watch how the widget tree of our current app is created.

Widget by widget.

The app codebase will be using is the counter app from my last tutorial regarding block to block communication,

you can find the source code for that in the video description.

Let's start with the beginning.

Every application must start at a point in Flutter.

This point is actually the required main function from inside the main dot dart file.

The run app method inside will start building the app.

Therefore our widget tree.

Every tree must always have a root node, a node where everything starts and develops from.

Therefore, every widget tree must always have a root widget.

In our case, the root widget is actually the first widget created.

More specifically the My app widget.

As I previously said, since every flutter widget is built within a build context, the My app widget

will also be built within a build context.

Now please pay a little bit more attention because I'll tell you the relationship between different

build contexts.

I'm going to tell you three completely wrong assumptions you might make during the development of your

app.

Firstly, you might be tempted to say, since all these build context instances have the same name,

then that means we're talking of the same build context over and over again.

Secondly, you might be tempted to say that since there is no build context instance mentioned as a

parameter to a widget, then the widget isn't built within a build context.

Last but not least, you might be tempted to think that the context in which the next child widget will

be built won't be related at all to the previous context.

Inside the widget tree.

All of these assumptions are terribly wrong and let me explain you why in detail.

We'll start with the first wrong assumption.

Even though the names of the build context instances are the same here and here and here, this doesn't

mean they are identical by any means.
This context is completely different from this one and completely different from this one.
This is crucial to understand.
Context in this case is just a name for a build context instance which is passed as an argument to different functions.
It's like having three different functions, both having the same named arguments, but when called the passed arguments, values will be different even though the arguments are named the same.
In this situation, when there is a build context context instance passed to a builder function, you need to start imagining the name context as being the build context in which the current widget is being created.
Therefore, in order to make things even more understandable for learning purposes, I'll rename these.
Context.
The following.
The context in which the My app widget is built will be called.
My app Context.
The context in which the block provider of type internet widget will be created will be called internet context.
And finally, the context in which the block provider of type counter widget will be created will be called counter qubit context.
I rename them this way so that you'll see better that they are completely different.
When you fully understand this, you will find it easier to name all of them contexts since you'll know which context goes where immediately.
Now that we figured out the first wrong assumption, let's move to the second one.
As I told you before, every widget inside Flutter has a build function which will require a build context instance in order for it to be built and most importantly, placed correctly inside the widget tree.
So if you're seeing the material app here alone, this doesn't mean it was built without any context.
It has an anonymous build context, but it's still a build context like every other widget and will be placed correctly inside the widget tree.
Finally, to discuss the last wrong assumption regarding the relationship between build contexts, you need to know that the build context of a widget keeps track only of its direct parent and nothing else.
So in our case, our widget build within its next context only knows that its parent is the widget built on the previous context and nothing else.

So to sum up the relationship between build context is a bottom up relationship because the only way to know what's at the top of the tree is to start from a bottom child and work your way up asking at each step for every parent that is above until you reach the top.

Build context only cares about its parent context.

It won't record any information about its child or children.

This is crucial to understand because this way you will know that the only way to move inside the widget tree is by choosing a context and then navigating up from there.

The only way is up.

No left, no right.

No down, just up.

This is the only way.

Build context.

Navigate between each other.

Coming back to our widget tree.

Every time you see the build function returning a widget, that widget is going to be a child of your current widget.

In our case, the my app returns in its build function a multiblock provider.

So this widget is going to be built in a new context this time.

Anonymous.

Therefore the Multiblock provider widget will be the child of the My app widget inside the widget tree.

Now to continue, we need to understand what Multiblock provider really does.

Remember from my third Flutter Block Concepts tutorial when I talked about Multiblock provider, I mentioned

that this widget was built to facilitate providing multiple blocks or qubits so that instead of nesting them one inside another, you can create a list and mention them inside it.

Need to understand that behind the scenes Multiblock provider will be still nesting.

Each block provider from the list.

Multiblock provider is just a way of making the code look more readable and organized.

So having this in mind, we can deduce that the block provider of type internet qubit is going to be the next widget in our widget tree with its own internet qubit context build context.

Then in a new counter qubit context build context, another child will be linked to our widget tree, the block provider of type counter qubit.

Let's continue further.

Whenever you see a child parameter, the widget passed will be a child of your current widget.

So material app goes up next inside the widget tree of our app.

Be aware that team data class is not a widget since it does not extend any type of widget class.

So we won't take it into consideration inside our widget tree.

In my sixth tutorial from the series regarding block access and navigation, one of the routing options we tackled was the generated routing.

So by default, our application will push in the default home route into the navigation stack, which is linked to our home screen widget.

Therefore, the next widget inside our widget tree is going to be the home screen widget.

Again, this widget comes with a build method and a new context which will call home screen context for learning purposes.

The return widget is the block listener, which will be placed right down the widget tree.

As we have previously done.

Its build context will be called Internet qubit Listener Context.

The block listener also comes with a child parameter the scaffold.

So this will be the next widget inside our widget tree.

Now please pay a little bit more attention.

As you can see, the scaffold in our case we.

Receives two widget parameters the app bar widget inside the app bar parameter and the center widget inside the body parameter.

When you see this kind of scenario with multiple parameters filled with different widgets inside the same widget, you need to know that these will be children of the same parent widget.

So in our case we have two widgets building from the scaffold widget.

One will be the child widget and the other one the center child widget.

We'll need to see for each of these two widgets which are their children.

In the case of the app bar widget, there is only one child, the text widget.

On the other hand, the center widget has a column widget child with multiple children.

Those will be again direct children of the column widget so they'll be on the same level inside the widget tree.

Let's write them one by one.

The block builder, the divider, the block consumer, the material button and again the other material

button.

The block builder returns text widgets within a new counter qubit builder context.

Depending on which state is receiving block, consumer is almost the same scenario.

It will return the counter text widget within similar naming style contexts.

Now, please pay more attention to what I'm going to introduce right now inside the material button.

You might remember from my tutorial regarding block routing and navigation that we have an onpress function

from where we're calling Navigator Dot Push named to push a new screen inside the navigation stack.

This is the moment when you will finally encounter why it is crucial to understand how build context

works inside Flutter.

You might think that this line of code is simply a default way of navigating inside Flutter.

Well, it's not like that.

Let me translate it to you.

Navigator dot of context Dot push named is equal to search for the closest navigator widget inside the

widget tree starting from the past built context instance and then after the navigator widget is found,

add the new screen widget as a child of the navigator widget.

So let's take this easily one piece at a time.

Search for the closest navigator widget inside the widget tree.

This means Flutter is going to do a lookup after the closest navigation widget starting from the past

Build context instance.

As you can see, there is a build context instance passed inside of Navigator State.

This is done to indicate flutter from which exact context inside the widget tree.

The lookup should start.

As I told you before, in order to find a specific widget in the widget tree, you need to choose a

starting context and then proceed upwards parent by parent until you find it.

So in our case, Flutter will start searching for a navigator widget right from the provided build context

instance, which in our case should be the context of the material button widget.

The thing is that since the material button is built within an anonymous context, we can't access it,

so we cannot mention it inside the navigator, that line of code.

But can we transform an anonymous context of a widget into a normal, accessible one?

Yes, we can, and this is done by wrapping the widget inside a builder widget.

The builder widget was created to be able to specify a build context instance name in which a widget

should be built.

So in our case, all we need to do is wrap the material button inside the builder widget and specify the name of material button context for the build context instance in which the widget should be created.

Now we can pass the material button context to the navigator dot of line of code.

For the second scenario, when we want to start searching for the navigator widget from inside the context of the home screen, we'll need to pass the home screen context instance to the navigator that of line of code.

So we'll have two scenarios to test.

One starting from the context of material button widget and the other one starting from the context of the home screen widget.

Okay, so let's simulate how Flutter works with both scenarios.

In the first scenario, we want to push the second screen onto the navigation stack right after the navigator widget is found inside the widget tree.

We're starting the lookup from the context in which the material button was created, which is the material button context, and then Flutter will ask for the parent widget of it.

It will go right to the column widget, then to center scaffold and block listener widgets.

Finally, Flutter will search inside the homescreen context for the navigator widget.

None was found there, so it moves again to the parent widget of the home screen widget, which is the material app widget.

Even though there is no written navigator widget inside our material app widget, you need to know that.

Material app has it inside its core functionality.

This is where the navigator feature starts, so if you are always wondering why the material app widget contains all the parameters for setting up the preferred routing option, well now you know why.

Because it contains the Navigator widget core which manages the navigation feature of your app.

Okay, so right now Flutter found the navigator widget it was looking for.

Now it's time for our navigator to push the second screen onto the stack.

But remember, it will push it from the context of the navigator widget, and that's the context of the material app.

So the new second screen widget will be placed as a child of the material app widget.

So the new widget tree following our previous rules, will transform into this.

Note that if you click the back button, the second screen widget will pop from the navigation stack and will no longer exist inside the widget tree.

In the second scenario, the same procedure applies inside Flutter, the only difference being that the lookup starts from the home screen context instead of the material button context, and that instead of building the second screen as a child of the material app will build the third screen.

You need to understand that the translation of the navigator dot of context line of code works Also when you are writing block provider dot of context, team dot of context, scaffold dot of context or any similar sequences.

Therefore block provider that of context means that it will search for a specific provided instance of a block or qubit inside the widget tree, starting from the provided context, since there might be multiple provided blocks or qubits.

That's why we're mentioning the type of it between angle brackets.

I hope you understood how built contexts work and interact with each other overall.

If you have ever wondered why wrapping the material app with block providers caused the blocks or qubits to be provided globally to the entire widget tree, now you know why?

It's because all the screen widgets from inside your app will be children of the material app widget, therefore creating and providing them before the context in which the material app is created.

Which by the way contains also the navigator widget means that whenever a new screen is pushed searching for the block or qubit unique instance by calling block provider dot of context from the context of that screen will go and search for the instance until it finds it above the material app widget.

That's also why if we wrap the home screen widget inside the block provider and then we push a new screen widget, which is going to be a child of the material app, you won't be able to access the block or qubit instance provided from the home screen inside that new screen widget.

Why?

Because this is the nature of how built context works.

Searching for a qubit or block instance starting from the context of the new screen widget will drive Flutter to the material app.

Since that's the parent of the new screen, then Flutter will go to my app root widget, and since it wasn't able to find any instance of the block or qubit down the road, it will throw the well-known error of block provider.

That all fails to find the context containing the specific block or qubit.

Another thing you need to understand is that you cannot access a block or qubit instance from the same

context in which it was provided.

So you must ensure block provider that is called within a child build context To analyze this theory in more detail.

Here's three examples you need to understand.

So first example analyze this code for a bit and let's apply what we learned so far.

Can you access the unique instance of Block A in this specific case?

Pause the video for a moment and think of it, then come back for the full explanation.

Okay, so the answer to this example is yes.

You can access the unique instance of block A by using block provider of block A of context.

Now let me explain to you why the first widget is built in a context Build context.

Then its child.

The block provider is built in an anonymous context right below it.

Bear in mind whenever you see an underscore passed as a parameter to a function, it translates to it does matter which parameter it's passed there.

We don't care about its name.

So in our case, we don't actually care about the name of the context in which block A is created.

Then we have a child called My Child, which is also built in a context build context.

Remember that even though these built functions have the same parameter names, they are not the same.

This is a completely separate context from the context above.

We also have a raised button built with its own anonymous context At the end of the widget tree in which we're asking Flutter to find us the first block provider of type block A So the context from.

Which will start searching is, as you can see, the context of the My child widget will start going up from there and voila, my app's parent is actually what we were searching for, the block provider of Block A, which contains the block, a unique instance success.

Let's move to the second example.

Here's the code again.

I want you to take a look and think whether the unique instance of Block A is accessible or not.

So what do you say if you think the Block A is accessible, there you are wrong.

And let's investigate why we have a widget built in a context built context at the top.

Then right below it, a block provider will create the block A into an anonymous context.

The raised button will also be built as a child within an anonymous context.

We're asking Flutter to find the closest block provider of type Block A starting with the past context.

If you pay attention, the past context is actually the one in which the top widget was built, since the other created widgets have anonymous context, which we cannot access.

So in this case, Flutter won't be able to find the block provider of type block A therefore throwing the well known error of block provider that of fails to find a block or qubit.

So do you think there is a way on how we could adapt this specific example in order to make it work?

The widget tree looks rather good.

If we only had an accessible context in order to start a lookup from there, we can now start a lookup from this context and since the block provider is actually above this context, we know for sure that the unique instance of the Block A is accessible.

This was the example number three, and I hope you understood why knowing how built context works is crucial.

We're almost at the end of this important tutorial.

What's left for now is for me to tell you a trick on how you can easily visualize the widget tree of your app without having to manually write it down every time.

So first of all, inside vs code you need to build and run your application, then hit control, shift P and search for dart dev tools, then open it in a new browser tab.

If you go to Flutter Inspector, you can notice the entire widget tree which is currently built inside your app.

As you can see, it's actually the same widget tree we manually built.

How convenient.

So whenever you have doubts in which context a specific widget is built, you can now double check it inside.

Flutter Inspector.

This was definitely a long but really, really, really important tutorial.

Build context is actually one of the key concepts of Flutter, so understanding how it works underneath will win you big time in the long run.

We have successfully covered about two thirds of this amazing flutter block course.

Congratulations.

In the next chapter you will learn what are the new additions of Flutter Block version 6.1.0.

It's a really nice update, so make sure you check it out.

With that being said, I'm looking forward to seeing each and every one of you in the next chapter.

Stay tuned and let's get Wicked.

9_comparing-objects-and-equatable-package

1. State not updating issue and how to fix it

Hey, what is going on, everyone?

I'm wicked.

And today I'm going to address an issue some of the new developers often encounter while working with the block library.

You might find yourself at a moment in time in which your block or qubit state won't update at all.

So in this tutorial you'll find out why this happens and exactly how we can fix it.

We'll also take a more in-depth look over the Equatable package so we know from previous tutorials that each block or qubit incorporates a stream of outgoing states.

Our state not updating problem translates into a new state from a block or qubit can't be emitted to the stream of outgoing states.

The cause of this problem is pretty simple.

You're probably emitting the same exact state as the previous one.

The problem with this is that block library does not allow two identical states to be emitted, one after another.

You can imagine this rule exists because of optimization techniques.

Why would you want to emit to identical states one after another?

After all, you probably don't want that.

But the code you wrote, unfortunately, translates into that.

First and foremost, you need to understand how Dart compares to objects, since that's how it also compares to states in order to see if they're equal or not.

From the documentation, we find out that two objects are equal if and only if both are the same object.

Unfortunately, this may not seem really straightforward and some people misinterpret it.

So let's say we have a myclass class containing an integer value.

Inside the main function, we'll create two instances of this Myclass object, both having the value set to one based on the definition.

What do you think this print command will display?

You might be tempted to say that they're equal, but they're definitely not.

What the definition is trying to say is that the only way we can get through here is when we compare A to A and B to B, Our A and B variables are not the same object, even though it looks like they are from the code, they are different.

Therefore, comparing A to B will return a false value.

So that's how the default equality operator works inside Dart.

But obviously we can override it to compare objects as we want to.

We want to compare these two instances to see if they're both instantiated from the same type of class and also to see if they have the same value stored inside them.

To do that, we can override the equal operator by pressing the control plus period keys right next to the class and then by choosing to generate equality.

So what this does is check if the other object we're comparing to is of type Myclass and if the other objects value is equal to the ones we're comparing to.

Now, as you can see, if we run this program, the print sequence displays true.

The question now is do we have to override this equal operator every time we're doing this?

Fortunately, no.

There's already a really popular Dart package which simplifies things a lot.

Its name is Equatable, so let's import this package into the Pubspec.yaml file and see how it works.

All we need to do now is to extend our class from the Equatable package and override its props.

The props will be a list of all the variables we would want Equatable to compare in order to decide the equality of two objects.

In our case, we want to pass the value variable to the list of props.

If we save and restart our project, you will see that it works exactly as if we would manually overriding the equality operator.

Let's test it even further.

What if we have different values?

It will return false.

But let's say we forget to mention the value parameter inside the props file.

As you can see, if we compare A and B, we'll still get true even though the values are different.

And that's because Equatable will only compare the run type of the objects.

Both of them are instances of Myclass class, thus marking them equal.

That's why it's important to pay attention to every scenario on how Dart compares objects.

Remember that each state from inside your blocks or qubits is still a class.

Nevertheless.

So everything we discussed applies to every state too.

Coming back to our state not updating issue.

We said that the cause of it is mainly the fact that the state we want to emit is actually equal to the previous emitted one.

In order to understand this in more detail, I have created a settings page containing two switch list styles, one serving for an eventual app notification feature and the other one serving for an eventual email notification feature.

So each of these two styles can be either on or off, depending on the user's preference to receive app or email notifications to manage this page.

I have also created a settings qubit, a qubit which will emit a single setting state Each time one of the tiles are toggled.

The settings state class will contain two boolean class.

App notifications and email notifications.

By default, the initial state of the qubit will be setting state with both of the fields set to false.

I have also created two functions inside the qubit.

One is triggered when the app notification style is pressed and the other one is triggered when the email notification style is pressed.

Both of these two functions will emit a single state according to what the new value of the boolean values will be inside the UI.

Of course, I use block builder in order to rebuild the switch list styles according to the specific values found inside the settings state.

I have also added a block listener at the top in which I'm showing a snack bar to see that the values of the settings state correspond to what's shown on the screen.

The settings qubit is provided globally above the material app widget, thus being available to all of our screens.

Remember, if you have doubts or don't understand a line of code, you can find the source code for this tutorial in the description and you can also leave me a comment right below the video.

Okay, now let's investigate the mistakes some of the users of block library make while coding mistakes that lead to the state not being updated anymore.

Let's switch our view to the toggle app Notifications method Inside Settings Qubit class.

I said that inside this function we'll have to retrieve the new value of the switch tile from the UI,

then emit a setting state containing the app notifications field populated with the new value.

Some people may do the following.

They would take the current state, which is a setting state, modify its app notification value directly

to the new value, subsequently emitting that state again into the stream of emitted states.

These two lines of code are really, really, really wrong and must be avoided at all costs.

First of all, we're mutating an existing state of our qubit.

This is a complete violation of the main principle of Bloch.

Remember, for every interaction the user makes with the UI, there should be a new state emitted from

the Bloch or qubit.

So you should never but never modify or mutate any already existing state from inside a block or qubit.

You can access its value using the state operator right here, but you should never modify it.

The biggest problem isn't, however, this line of code, but rather the next one.

It may seem that if you modify the app notifications value above with the new value the state were emitting

here will be different from the previous one.

This is again wrong.

Remember that we discussed earlier on how Dart compares objects at the moment.

Currently this emitted state is to dart the same exact state as it was before.

It doesn't matter if we modified its value here.

It's just like we previously compared A to A and B to B, it will still return.

True, since it's the same object due to the previous state being equal to one you're trying to emit.

Block library won't emit a new state to the stream of states.

Therefore not updating the UI.

This can be noticed if we save and run the project.

Right now.

As you can see, if we click the switch tile, nothing happens.

In order to fix this, we need to erase these lines of code.

I don't know if you noticed, but inside the Settings state class, I have this copy with method I generated

by clicking control plus period next to the class name.

If you're not familiar with this method, what it essentially does is create a new instance of a setting

state by copying the entire previous one.

And then inside the methods parameters, you can mention which field you want to modify.

After it gets created in our class, we can emit a new state by calling state dot copy with which will create a new settings state and copy the content from the previous one to the new state by also setting up the new value of app notifications.

As you can see, if we save and run the project right now, the application is going to work perfectly.

As I told you before, another mistake users do is that whenever they use the equatable package, they forget to place the values inside the props variable.

In our case, if we were to do that, we would have set our settings state to extend Equatable.

Then let's say we forgot to write the app notifications variable inside the props.

This means that Dart will only compare to settings state by checking only the values of email notifications variables.

So for example, if only the app notifications variable modifies Dart won't care about it and will still return true while comparing the two states.

So if we save and run this project and click the switch style for the app notifications feature, it won't do anything since Dart thinks it's the same state as before and won't emit it.

So please pay attention while working with Equatable package and with how Dart compares objects.

In general, we are easily but steadily approaching the end of this amazing course.

Hope you really liked it.

Until now, only two chapters separated us from the end in.

The next chapter, I will teach you a really important topic every blog developer should know.

That is how the state of an app can be maintained and restored from the app's internal storage.

Hydrated block is the key to that.

With that being said, I'm looking forward to seeing each and every one of you in the next chapter.

Stay tuned and let's get Wicked.

10_local-storage-with-hydrated-bloc

1. Local Storage with Hydrated BLoC

What is going on, everyone?

I'm wicked.

And today I'm going to show you how easy it is to maintain the state of your block build application with dehydrated block package.

This is a topic that is often omitted, but to be honest, I think it's one of the most important pillars of a flutter application.

By default, every application you'll build inside Flutter won't be able to maintain its state after rebooting.

As an example, we'll take a look at our current counter app.

If we increment the counter to five and then hit the restart button, the state of the application will revert back to its initial state.

So in order to avoid this, we'll need to somehow store the entire state of our application so that whenever the application reboots, it will be able to access the already stored data.

Okay, so states we've all been talking about them since the beginning of this tutorial series.

We learn that a state is a group of data that is stored inside the application at a moment in time.

Right.

Remember, BLOCK Library is a powerful state management solution.

BLOCK Library helped us create the state, but now it's our duty to help its store, therefore maintain them.

The way we're going to do this is by using the hydrated block package.

In order to understand how hydrated block works, we need to see where and more importantly, how it stores the data of our app.

So where are we going to store all of our data?

This place needs to be fast and close to the application.

Therefore, the most intuitive solution would be actually the local storage of the device the application runs on.

You may have heard that every app running on an Android device will benefit from its own internal storage

where it can store some of its data.

So this is where hydrogen block will also have access to store our application state.

Let's start with this.

First, I'll open up my previous project Inside the Scope, a project that can be found on my GitHub page, which is listed into description.

Then we'll need to add dehydrated block dependency.

So we'll open up the box, pick that YAML file and list it.

Right there will also need a path provider dependency because this will help flutter retrieve the path for the storage of different devices the application may be installed on.

Now we'll let you tell a hydrated block what will be the accessible storage area required to store our data onto.

To do that inside the main function from our main dart file, we need to set up its storage.

This hydrated build function asynchronously creates the connection from the hydrated block to the storage on where it's going to store the data by default.

If we're not mentioning anything to the storage directory parameter of the function, the data will be stored to a temporary storage which can be easily removed by the operating system at any time.

So it's not really recommended to store anything there.

Instead, we could set the storage to the one provided by Android for the current app.

This entire line of code will call native code to initialize the required storage and link it to hydrated block.

But in order to do that, we need to initialize the widget bindings before we start building our app.

Otherwise we wouldn't be able to call the native code, so keep that in mind.

Okay.

So currently we establish the connection between the hydrated block and the app storage layer.

Now all we need to find out is how hydrated block will interpret and save the data.

We want to be stored inside that storage layer.

And the answer is Jasons.

You must have heard about Jason before.

I talked about them in my full tutorial in block architecture when I say that the data fetched from the outside of the app may come in adjacent file format, so similar to how the app can import the data from adjacent format, the same thing can happen vice versa.

The application can also export the data to a JSON format.

What's left for us right now is to decide which data we want to save in the storage.

So let's start by maintaining the state of the counter qubit.

The state of the counter qubit is found in the counter state class.

So let's code that first.

As I told you a couple of moments ago, we'll need to export the data of this class into a JSON later on.

Hydrated block will take the exported JSON and store it safely into the storage.

You can easily create the to JSON and from JSON methods of an already existing class by clicking control plus period and then by selecting JSON serialization.

This is only available through the dark data class generated via Xcode module.

I know a lot of you asked me what this module was, so here it is installed on my VTS code.

As you can see, the module generated all the functions we needed for our counter state class implementation.

Now I want you to understand why it created four methods to map from map to JSON and from JSON.

A JSON is a file having the json.

An extension appended to its name.

This is mostly how an API will reply to your HTTP request.

Dart language can only understand its own types of data like integers, doubles, booleans, maps, lists and so on and so forth.

It doesn't have a special type called JSON, so we need to take the JSON file, which is mainly a really large string and decoded to a type that is familiar with the type that is similar to the structure of a JSON.

Is the map similar to a JSON?

A map keeps multiple key value pairs inside it.

So we have the JSON file.

We'll decode it into a map variable using the from JSON method.

Then we'll use the map to create a class instance with the from map method.

Vice versa.

If you have an instance of a class with some data inside, it will use that to map method to create a map out of that specific instance, then use the to JSON method to create a JSON file out of that map.

And that's the entire process of converting an instance of a class to a completely storable JSON and vice versa.

That's the purpose of all these methods, and that's how helpful this dark data class generator function is.

Imagine how much time it would take to manually write this function.

So at this point, the counter state class can be fully exported and imported to and from JSON.

Now we want to find out where and when hydrated block will come into action.

So let's switch our attention to the counter qubit class for a block or qubit to become hydrated.

That is for the state of a block or qubit to be saved and retrieved from the storage.

We need to use the high rate in mixing to the class.

Note that we can also use the hydrated qubit or hydrogen block extension of the counter state class instead of using the hydrogen mixing.

As you can see, that warns us that we need to override the two methods of the hydrated mixing from JSON and to JSON.

These two methods are the pillars of storing and saving the state to the storage while also being able to retrieve it back when the app is rebooted.

Take in mind also that these methods should have been called from map and to map, since all they do is import and export a map type from the storage.

Since the JSON is most of the time translated into a map, it is acceptable to call them from JSON to JSON.

But you have to keep in mind the difference.

All we have to do now is to implement those functions.

You need to take in mind that the to JSON function is called for every state emitted by the counter qubit and that the from JSON function is called every time the application needs the locally stored data.

Let's go the to JSON method first.

So every time there is a new counter state emitted with the new counter value, we want to save its data to a map and then send it to the hydrated block to store it onto the local storage.

When we want the app to access the last safe data from the local storage, high rated block will call the from JSON function and retrieve the JSON, which is already converted into a map.

As I told you before, we need to return a new instance of the counter state populated with the data from that map.

So we'll use the counter state from map method for that to happen.

If we finally save a run the project now you'll see that after we increment the value to five and then restart the application, the value will still remain five.

And that's the exact behavior that we wanted to understand this tutorial much more in depth.

Let me get the whiteboard one more time and simulate every step of the process.

So we have the app, the counter cube with its hydrating block plugin attached, the counter state and our storage components.

We have a counter value of zero, the initial value of the counter qubit.

When a user presses the plus button, the increment function from inside the qubit is triggered, causing the counter cube to emit a new counter state with an incremented value of one.

Here, the hydrated block is notified that a new state has been dispatched and calls the to JSON function.

The to JSON function will call the two map method from inside the counter state, which takes all the attributes of the class and their values and puts them into a map.

So right now we'll have a map which looks exactly like this.

This map is then being taken care of by the hydrated block plugin, therefore being stored on to the storage.

Now let's say the app got killed or has been restarted by the user when it will be opened again.

High rated block will retrieve the last known data from the storage as a map of data.

Then, with the help of from JSON method of counter qubit and with the help of from map method of counter

state, it will create a new instance of counter state with the previously stored values, thus ending the cycle and maintaining the counter state.

You might be wondering, well, what if we have a ton of read and write operations from the storage?

Won't that slow down the application?

No.

Hydrogen block is not only a real.

A useful plugin, but it's also really optimized and that's thanks to the high package that it used under the hood to store the data.

Hive is the most efficient way of storing data.

Take a look at these benchmarks.

It's absolutely blazing fast.

So to sum up, every state from your blocks or qubits can be maintained and saved to the storage.

All you need to do is to decide which ones you want to keep in storage and which ones you don't.

For example, inside this application, you may store the state of the counter qubit and the state of the settings qubit.

But you won't need to store the state of internet qubits since the state of the Internet connection must be retrieved constantly, even when you reboot the app.

This was chapter number 11.

You are one chapter away from becoming a block hero.

Isn't it exciting?

In the final chapter, we're going to buckle up everything we learned and play with all the debugging methods of a block or qubit.

So in case something goes wrong with them, we'll be able to successfully track and debug them at a glance.

With that being said, I'm looking forward to seeing each and every one of you in the last chapter of this course.

Stay tuned and let's get wicked.

11_debugging-bloc

1. Debugging BLoC

Hey, what is going on, everyone?

I'm wicked.

And today I'm really happy to present you the last chapter of this amazing Flutter blog.

Complete course.

In this chapter, I'll introduce you to debugging blocks and qubits code facilitating vs code extensions and some of the most important naming conventions you should follow when working with block library on a large project.

Okay, so until now, we kind of understood in big words how Block library works.

What are the main concepts and how to structure the code of a block driven app?

But what if something is wrong with one of our qubits or blocks?

Is there a way we can specifically debug them?

Yes, there is.

Both blocks and qubits have specific debugging methods you can override so that you can observe their behavior while the app is running.

Remember that a block is just an advanced type of qubit block extends qubit.

Therefore, every debugging method found inside the qubit will also be available inside a block at the same time, a block will benefit from some separate specific functions, thanks to its special ability of being triggered by a stream of events.

So let's start with the qubit first.

Remember this diagram we discussed way back then?

When a qubit emits a new state, we say that a change occurs.

Therefore, we can deduce that a change contains only two pieces of information the current state and the next state.

But why are they called current and next state instead of current and previous state?

Well, you need to understand that this own change method is called right before the state will be emitted down the stream of states.

So at that moment in time, it makes more sense to call them current and next state.

So to sum up with the help of this overridden on change function, we can observe all changes for a given qubit.

In order to see this in action, I'll open up my last project inside vs code.

You can find it on my GitHub page as always in the video description.

Then I'll simply navigate to the counter qubit and override the onchange function.

As you may notice, this function takes a change instance of type counter state parameter.

So every time a new state is emitted down the stream of states, this function will be called with a change instance containing both the current and the next state.

Obviously this will help you track and debug the stream of emitted states, which is really important.

Inside this function you may adopt two approaches.

You can either manually code what you want to be printed by accessing both counter values of the current and previous state, or you can opt for a much faster and efficient approach.

You can simply print the passed change instance as a parameter to the onchange function.

But how does Dart know how to print an instance of the counter state, you may ask.

Well, at the moment it doesn't, but if we go to the counter state class, we can override the two string function which is mainly called when Dart wants a string representation of the object.

Instantiated from the counter state class.

This is exactly what we want.

If we save and run the project right now, you'll notice how simple it is to track and debug a qubit whenever a new state is emitted by the counter qubit, either by incrementing or decrementing the value.

A new change is printed into the debug console, letting us know what the current state is and what the next state will be.

Another important method which you can override inside a qubit is the Onerror method.

This is triggered whenever there is an exception inside a qubit.

An application is most likely to have some exceptions during its lifetime.

We never rewrite something to the internal storage.

Something may go wrong with the operation.

If something goes wrong, then we may want to dispatch an exception with a specific error message.

We can do that by using the add error function from inside any qubit.

This function takes in two arguments the exception you want to dispatch and the stack trace to where the program ran until it retrieved that exception.

Let's add this function randomly inside the Tojson method to simulate something that may go wrong with

it.

So whenever the two Json method is called, that is, whenever a new state is being emitted by the qubit, a new error will be added.

At the same time, due to the fact that we overridden the Onerror method inside the qubit, the app will print the exception and its stack trace so that you'll be able to understand what went wrong with the application.

Okay, so we finished with how we can debug and trace a specific qubit.

But what about the block by extending qubit block is actually a more advanced type of qubit.

Comparing both of them.

We learned in the past that the only difference between a qubit and a block is that the block is event driven.

That means a block does something in response to a stream of events dispatched from the UI.

In comparison, a qubit does things whenever.

One of the predefined methods is called.

Therefore, due to the fact that block reacts as a response to a stream of events, there must be a way to debug and trace this stream of events to and indeed it is the first method that can be overridden, which is only available to a block is the on event method as his name is implying.

And similar to the on change method on event is simply a method that is called as soon as the new event gets dispatched to the stream of events.

Another block specific method that can be overridden is the on transition method.

On transition method is kind of a combination between on change and on event, since it displays the current state, the next state, and also the event that triggered the block to dispatch a new state over the current state.

So by calling on change, you'd get this output and by calling on transition, you'd also get this.

But in addition, you'll also retrieve the event that caused the states to change.

So to sum up, along with the on change and on error methods, methods borrowed from the qubit, a block also comes in addition with its specific ones like on event and on transition.

Now another important question will arise in a huge application.

You'd have maybe more than 2030 blocks and qubits altogether doing different tasks.

What if you'd want to track each and every one of them?

Will You have to manually overwrite every function inside every block or qubit.

Well, you could do that, but it will be obviously time consuming.

Instead, we'll use a block library internal feature which manages everything related to observing and debugging qubits and blocks.

I'm talking about the Block Observer.

The Block Observer is a utility class from Inside the Block library, which contains every previously mentioned methods like on change, on error, on event, on transition, and also two extra ones on create and one close.

This last ones are called whenever a block or qubit has been created or closed.

We'll take a look at those in a couple of moments in order to test block observer.

First and foremost, we need to erase the already overridden methods inside our counter qubit, of course, to avoid multiple identical outputs.

Remember, Block Observer will take care of every qubit and block inside your app.

Then let's create a new folder called Utility Inside our Logic folder, inside of which I'll create an app underscore block, underscore observer dot dart file.

Here I'll create an app block observer class that obviously extends the block observer managed internally by the Block library.

After that, I'll make sure to override every method we discussed today.

Most of them I won't be able to test since we're only playing with qubits at the moment.

But the key of this tutorial is to understand in big words what they're exactly doing.

So if we take a look at the Onchange method here, we can print the change exactly as we previously did in the tutorial.

This unchanged will react globally right now to every change from every qubit.

Therefore, the change parameter can be a type of counter state, internet, state or setting state.

We need to make sure that every one of them have their two string method overridden so that we can easily print their values.

Let's switch our view to the Oncreate and Onclose methods.

If you've ever been curious on how and most importantly, when our qubits get created inside our app, now it's the time to learn one important key factor.

So let's print whatever qubits gets created here and observe a really important difference.

One thing that's left to be done now is to navigate inside the Main.dart file and set up the block dot observer to our newly created app block observer instance.

If we save and run our app, we'll notice an interesting fact even though we have set up three block

providers to get three qubits provided globally, only two of them were created.

When we opened the app, the counter qubit and the internet qubit.

But what about the settings qubit?

Why wasn't it created?

Is there something wrong with it?

If we click the settings icon, we can observe that Right after we pressed it, the settings qubit got created.

So why exactly is this happening?

Well, let's switch our view to our block providers.

If you have ever wondered what these boolean lazy variable was designed for, now you'll understand

by default the lazy parameter of a block provider is set to true.

That means the provided block or qubit will be created only when the app really needs it.

In our case, the settings qubit was created when the app needed it in the settings page.

So this is a really cool optimization technique.

As a result, if we set the lazy parameter to false the settings qubit.

Will be created right after the app runs.

Another fact you may have observed is that the first change occurring in our app is actually the one where we're checking for the type of internet connection.

Then, whether we're incrementing or decrementing our counter value.

Other changes occur into the debug window.

Last but not least, if we modify the notification settings, we'll also see a change with its respective values.

So we're finally done discussing how we can debug and trace all of your blocks and qubits inside your app.

I hope you understand how everything works.

Now it's time for me to continue showing you some of the tips and tricks that will make your development life easier while working with Block Library, I'll talk a little bit about some of the most frequently asked questions I got from you guys.

First question How do you know when you should create a qubit or a block?

The general rule is that you should create a qubit or a block for every feature of your app.

As I said multiple times and as you noticed in my tutorials for every feature my app had, I created a new qubit.

Whenever we're talking about the counter feature, the internet feature or the settings feature.

So that's the main rule.

If you want your app to be able to login and create user accounts, then you should have an authentication,

qubit or block.

If your apps work with different models like countries, fruits, weather, planes.

ET cetera, then you should have a block or qubit for each and every one of them.

Then the next question would be When should I create a qubit and when should I pick a block instead?

The general rule of this is that you should start by designing a qubit for your feature.

And then if you need the advanced features of a block, you can easily refactor the code and switch

to it.

You know, the only difference between a block and a qubit is that a block is event driven.

So you must think in which scenarios of your app.

It will be better to have events triggering a block to do the job instead of having a method to trigger

a qubit to do its job.

Remember, qubit works with simple and plain methods.

It's not a stream of methods.

Block, on the other hand, works with a stream of events.

So you must take advantage of what operations you could perform on that stream.

Another question I frequently get is what's the naming conventions you should follow when designing

a block or qubit?

The answer here is not really a general rule.

You can name the blocks, qubits, events, methods and states.

Whatever you feel, it makes sense for your app.

There are, however, some naming conventions that will make your code look more organized and straightforward

for the states.

It's recommended that you name them after this pattern.

A block or qubit subject, plus the action plus the state of the action.

For example, if you have a weather block, some of the weather states may sound like weather fetch

success, weather, fetch in progress or weather fetch failure.

If you have a user qubit, some of the user states may sound like user create success user create in

progress, user delete failure, user fetch, success, etcetera.

You can see where I'm going with this for the events it recommended that you name them after this pattern

block or qubit subject plus the action.

The event itself in past tense events should be named in the past tense because events are things that

have already occurred from the block's perspective.

For example, if you have a user block, again, some of the user events may sound like user fetched,

user deleted, user created, user modified, etcetera.

Now what about the methods of a qubit?

Methods of a qubit should be named so that they denote exactly what a qubit is working on.

For example, if there is a user qubit, the methods inside it should sound like fetch, delete, create,

modify and so on and so forth.

As I told you, these are only recommended conventions.

Feel free to modify them as much as you want, as long as it makes sense inside your own app.

One last question I receive in the comments is what vs code extensions do I frequently use?

So here's a list of every vs code extension I use.

The first one is better comments.

I use this to highlight some of my code in a brighter color.

The block extension is a must when working with block library just because it offers you many time saving

options like wrapping widgets and useful code snippets.

Bracket pair Colorizer, on the other hand, is also an awesome extension.

I use to keep track of some of my brackets.

You'll find a dozen of brackets inside Dart, so this tool is really efficient.

Dart Extension adds support to VS code for the Dart programming language and provides tools for effectively

editing, refactoring, running and reloading apps.

Now, maybe the most popular one I use in my tutorials, the Dart Dataclass generator is one of the

best.

Stations you can have for first class cogeneration, like generating class constructors to string to

Json and from Json methods and many, many more.

Dart import is a plugin I often use to refactor and optimize my imports inside my app.

It's an amazing time saver Extension Flutter like the Dart extension is also a really must have plugin

for vs code, adding support for effectively editing, refactoring, running and reloading flutter mobile

apps as well as support for the Dart programming language.

Gradle Language support is a very useful extension to manage and code the gradle files found inside the Flutter project.

Now about the theming of VS code material, icon theme and vs code icons are the VS code extensions I use for personalizing my folder icons.

And last but not least, Pubspec assist is one of the most amazing extensions which will help you add the latest dart and flutter packages from pop dev right inside the pubspec.yaml file.

This is a plugin I really use a lot, so I definitely recommend it.

And I think this wraps out everything I wanted you to understand not only for this tutorial but also for this entire block from Zero to Hero tutorial series.

It is finally time to end this tutorial series.

I have spent so much time making every tutorial and I really hope I could help you in learning the really popular block library.

If I did, I would really appreciate it if you could support me by subscribing to this channel and by liking and sharing the entire tutorial series, we learned everything about Block, but now it's time to put everything to practice by building real world apps.

This will be the next tutorial series.

I invite each and every one of you to join.

Until then, as always, take care.

Stay safe.

Wake it is out.

Bye bye.