Maps & Routing (Refer App: chronograph)

Let’s continue our journey with flutter with a sort of ‘mini project’. This time, we’ll build an app which displays the world time. We can choose any country we want, and based on the time zone, the respective time for that country will be displayed.

Let’s get though with the barebones of our app. For that, we’ll create a new package, just in order to better organise things. We’ll call the package as ‘pages’. Our app will contain 3 pages:

1. The home page—which will display the time
2. The loading page—which will be displayed when the app is loading the time zones.
3. The choose\_location page—which allows the user to pick a location whose time is to be displayed.

SIDEBAR: In order to display content in such a way that it is not obstructed by the top notification bar (containing battery status, wifi, time,etc), we wrap our content with the ‘SafeZone’ widget.

We’ll create 3 basic dart files for this, which will just contain a basic scaffold, and text widget in order for us to identify these pages. Thus our app contains a total of 4 dart files:

1. Main.dart (Which is automatically created when we create the project)
2. Home.dart
3. Loading.dart
4. Location\_choose.dart

Our files currently look like:

1. Main.dart

**import 'package:flutter/material.dart'**;  
**import 'package:chronograph/pages/loading.dart'**;  
**import 'package:chronograph/pages/location\_choose.dart'**;  
**import 'package:chronograph/pages/home.dart'**;  
  
**void** main() => runApp(MaterialApp(  
 debugShowCheckedModeBanner: **false**,  
 home: Home(),  
));

1. Home.dart

**import 'package:flutter/material.dart'**;  
  
**class** Home **extends** StatefulWidget {  
 @override  
 \_HomeState createState() => \_HomeState();  
}  
  
**class** \_HomeState **extends** State<Home> {  
 @override  
 Widget build(BuildContext context) {  
 **return** Scaffold(  
 body: SafeArea(  
 child: Column(  
 children: <Widget>[

Text(‘Home screen’)  
   
 ],  
 )  
 ),  
 );  
 }  
}

1. Loading.dart

**import 'package:flutter/material.dart'**;  
  
  
**class** Loading **extends** StatefulWidget {  
 @override  
 \_LoadingState createState() => \_LoadingState();  
}  
  
**class** \_LoadingState **extends** State<Loading> {  
  
 @override  
 Widget build(BuildContext context) {  
 **return** Scaffold(  
 body: Text(**'Loading Screen'**),  
 );  
 }  
}

1. Location\_choose.dart

**import 'package:flutter/material.dart'**;  
  
**class** ChooseLocation **extends** StatefulWidget {  
 @override  
 \_ChooseLocationState createState() => \_ChooseLocationState();  
}  
  
**class** \_ChooseLocationState **extends** State<ChooseLocation> {  
 @override  
 Widget build(BuildContext context) {  
  
 **return** Scaffold(  
 backgroundColor: Colors.*grey*[200],  
 appBar: AppBar(  
 backgroundColor: Colors.*blue*[900],  
 title: Text(**'Choose a location'**),  
 centerTitle: **true**,  
 elevation: 0,  
 ),  
 body: Text(**'Choose Location Screen'**),  
 );  
 }  
}

Before we proceed any further, we need to know about ‘Maps’ in dart. A map in dart is the same as a HashMap in Java. It is a datatype that represents a collection of key value pairs separated by commas.

So we can write something like:

void main() {

Map student={'name':'Moe','age':8};

print(student['name']);

print(student['age']);

}

The output will be:

Moe

8

Notice the syntax here. In order to define the map, we need to specify the keys and values separated by a colon. Those elements which are strings are included inside inverted commas. In order to access the elements of the map, we need to specify the key within square brackets.

The reason we needed maps is because of ‘route mapping’ in flutter. Till now we dealt with apps with only one screen. But, in reality all apps have a lot of different screens. Our worldTime app also has 3 screens. In the java world, we knew them as ‘activities’ in android.

Thus, we need to specify a route tree to flutter so that it gets an idea of where exactly in the app is the user currently present i.e. which activity is the user currently in. We do so by specifying the route of the activities in a map. This map is called the ‘route-map’ and is specified in the routes property of the MaterialApp.

The routes property takes in a ‘map’. This map as stated earlier is a collection of key-value pairs. The ‘key’ in this map are the routes. These routes are nothing but strings. They have a format like the directory structure in windows.

e.g: ‘/’ is the home page. This is the first page that the user views when he launches the app.

‘/mycontactspage’ can stand for the contacts page. This specifies that whatever the value of this key is, it specifies the contacts page.

‘/about\_page’ can specify the about page.

Now, the values of these keys are functions. These functions take in the ‘context’ object as a parameter. Remember context from android java? Yuppers, this is the same context object. This object keeps track of the current state of the mobile device. It also records where in the routing tree the user currently is. This function returns what widget we want to load up once we get to that page or launch that activity.

So, for example, our home page route could be like:

**'/home'**:(context){**return** Home();}

Our route map looks like:

routes: {  
 **'/'**:(context){ **return** Loading();},  
 **'/home'**:(context){**return** Home();},  
 **'/location'**:(context){**return** ChooseLocation();}  
}

CAUTION:

When we specify the route tree, we should not include the home property. This is because the home property specifies what the user sees when he directly launches the app. But, we do this when we specify the ‘/’ route. Again specifying the home property will make the code redundant.

For our app, we’ll specify the loading screen as the launcher activity because we’ll need to display it while we load data into our app. But for testing purposes, if we do want to override the routes map, we use the initialroute property of the MaterialApp widget. As the value of the initialRoute, we specify the key of the route map. For example, if we want to override the ‘/’ key and want to make the Home route as the launcher route, we can write it like this:

initialRoute: **'/home'**

Our main.dart file looks like:

**import 'package:flutter/material.dart'**;  
**import 'package:chronograph/pages/loading.dart'**;  
**import 'package:chronograph/pages/location\_choose.dart'**;  
**import 'package:chronograph/pages/home.dart'**;  
  
**void** main() => runApp(MaterialApp(  
 debugShowCheckedModeBanner: **false**,  
 initialRoute: **'/home'**,  
 routes: {  
 **'/'**:(context){ **return** Loading();},  
 **'/home'**:(context){**return** Home();},  
 **'/location'**:(context){**return** ChooseLocation();}  
  
 },  
));

Now, we want a button in our home activity which when pressed navigates us to the choose\_location activity. So, we create a flatbutton.icon. In the onPressed property we call a function of the Navigator class called pushNamed. This takes in 2 arguments:

1. The context object
2. The route we want to push on top of the current screen

Navigator.pushNamed(context, ‘/location’).

Inside it, we write the route we want to go to. It essentially pushes the new screen on top of the current one. The current screen still exists underneath. (Kind of like starting intents without disposing the previous activity).

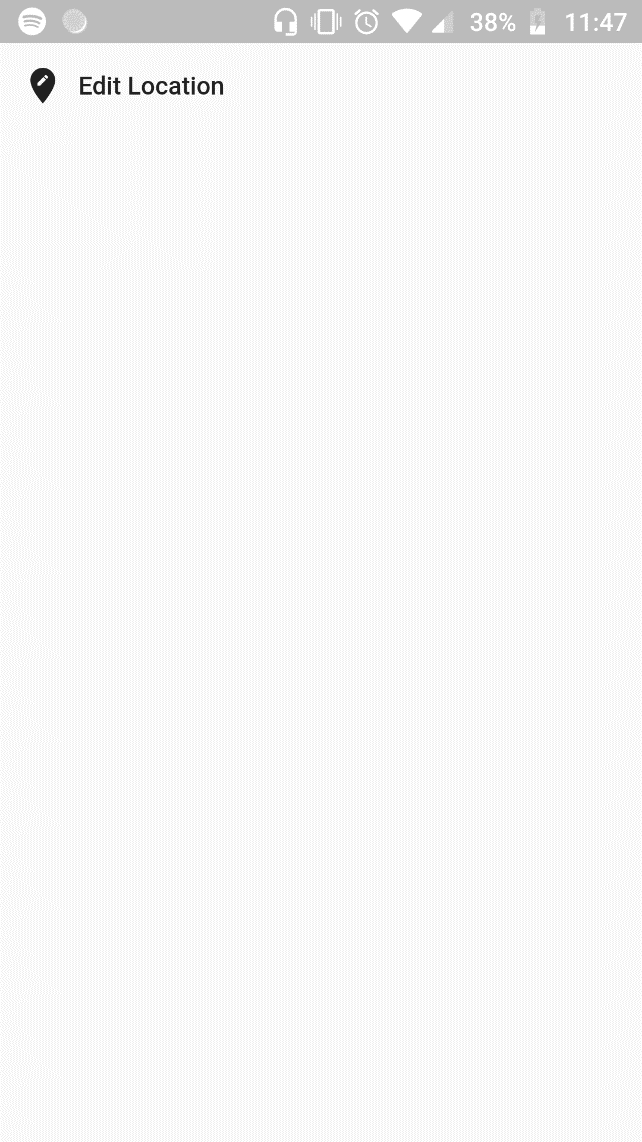
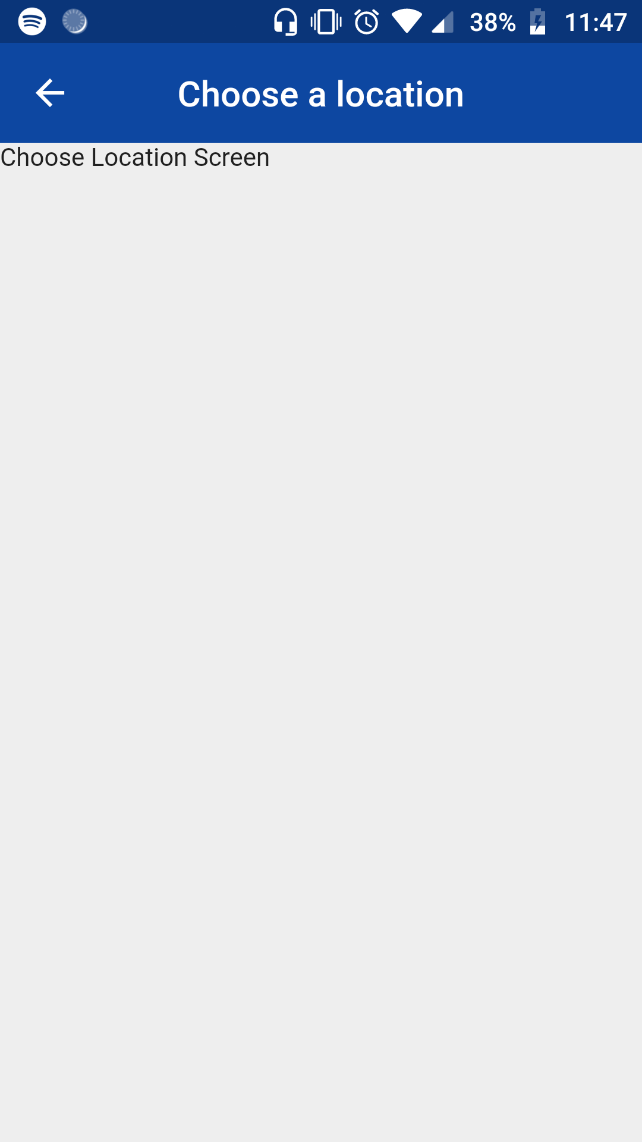
FlatButton.icon(  
 onPressed: (){  
 Navigator.*pushNamed*(context, **'/location'**);  
 },

Our home.dart file looks like:

**import 'package:flutter/material.dart'**;  
  
**class** Home **extends** StatefulWidget {  
 @override  
 \_HomeState createState() => \_HomeState();  
}  
  
**class** \_HomeState **extends** State<Home> {  
 @override  
 Widget build(BuildContext context) {  
 **return** Scaffold(  
 body: SafeArea(  
 child: Column(  
 children: <Widget>[  
 FlatButton.icon(  
 onPressed: (){  
 Navigator.*pushNamed*(context, **'/location'**);  
 },  
 icon: Icon(  
 Icons.*edit\_location* ),  
 label: Text(**'Edit Location'**)  
 )  
 ],  
 )  
 ),  
 );  
 }  
}

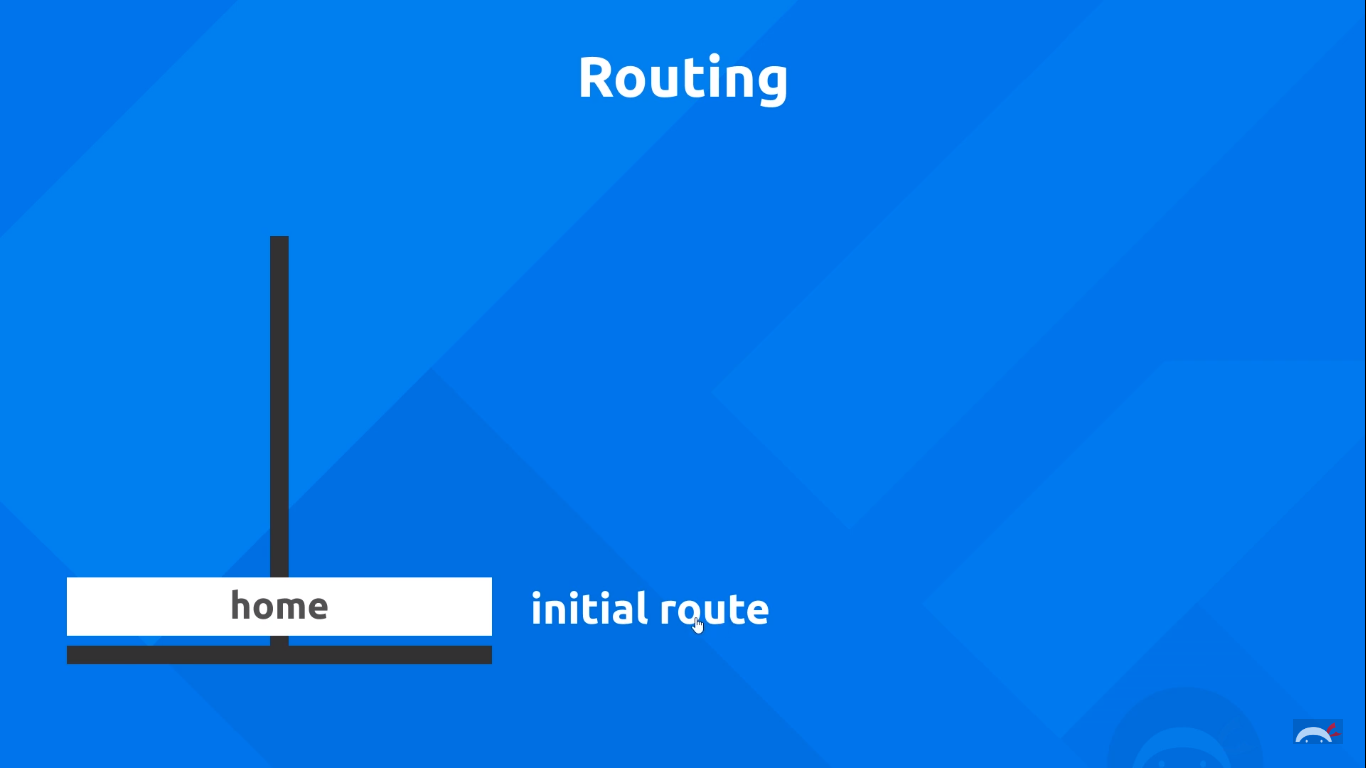
Now, a fun fact about material design in flutter, is that once we come to the choose location screen in our app, we see that it has an appbar.

BUT, when we arrive on the screen after navigating from another one, flutter AUTOMATICALLY adds a ‘back’ arrow to the appbar! Isn’t that just awesome?

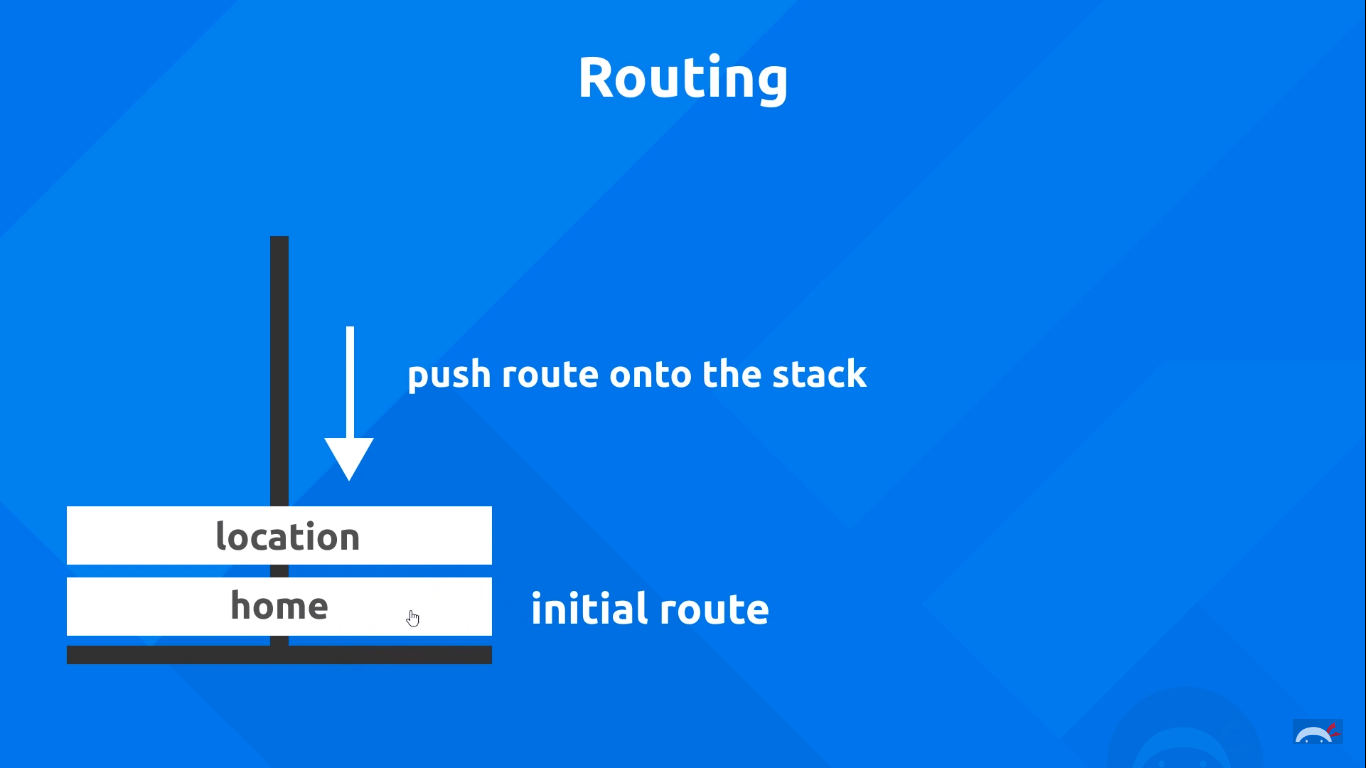
 

A little bit about routing:

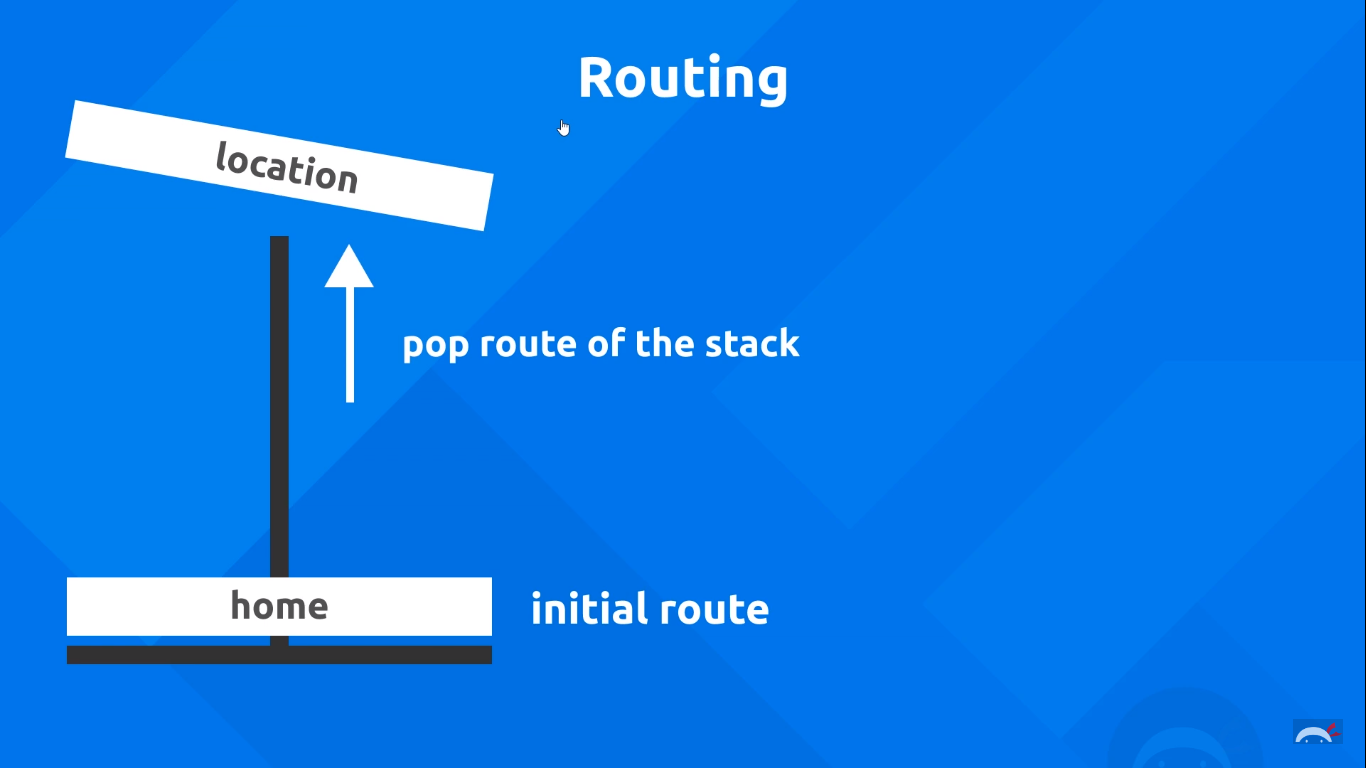
It is very important to understand the concept of routing between screens inside our flutter app. We can imagine our app as a stack of screens. When the user launches the app, there is only one route on the stack which is called the ‘initial route’.



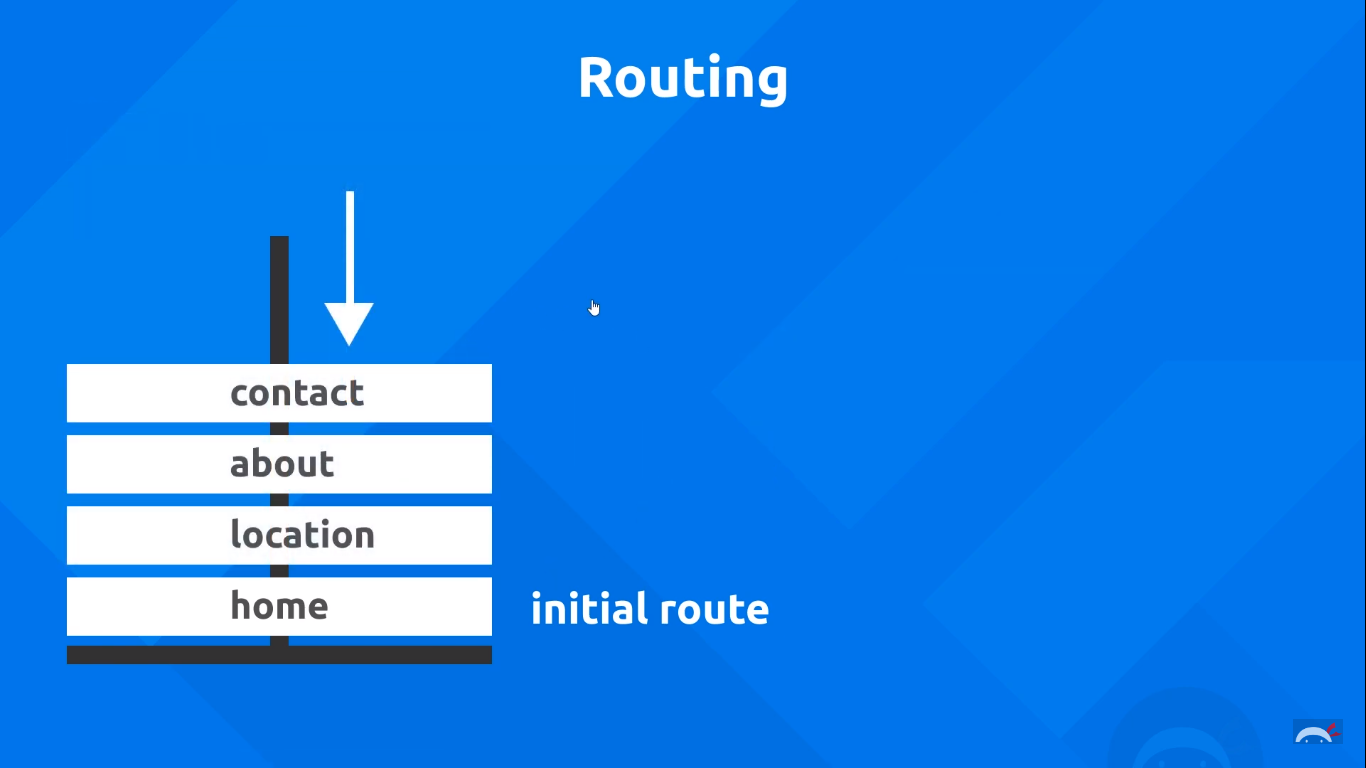
Whenever we use Navigator.push method, we essentially push a new screen on top of the already existing initial route. The old screen still sits underneath the new pushed screen.



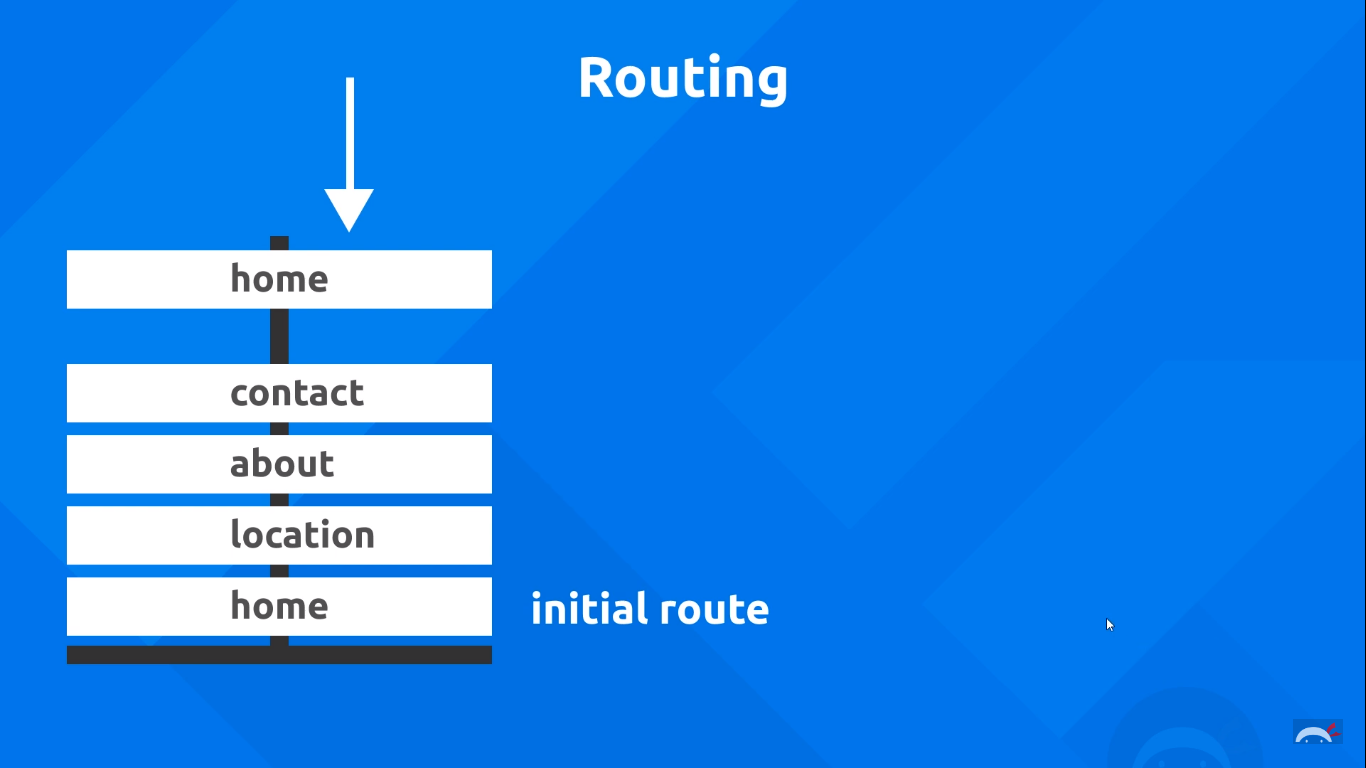
When we press the back button, it pops the current route from the stack so that we see the older route which was sitting underneath.



However if we are continuously pushing our routes one on top of each other, eventually we’re going to end up with a tall stack of routes.



This however becomes very tricky and difficult to manage in case of larger apps. So, imagine if we want to go back to the home screen of our apps. It is probably not a good idea to push the home screen again, as we already have it in our stack. It would be redundant.



We’ll be learning more about managing routes in upcoming lessons.

Widget Lifecycle:

Till now we have learnt about 2 kinds of widgets: Stateless and Stateful. A quick recap is shown below:



Stateful widgets have a number of different lifecycle methods that we can tap into. Let’s see some of them here:

initState() method: This is the first method which is called once our state object has been created.

Build() method: It runs quite a lot in a stateful widget as it needs to update the UI, and therefore build the widget tree.

Dispose() method: It is executed when the widget is removed from the app.



Asynchronous Code:

An asynchronous task is an action which starts now, and finishes at a particular point of time in the future. An example could be to interact with an API or a database to fetch data. It might take a second or two to complete the request, and therefore it is termed as asynchronous.

In the meantime however, our code should not stop until that request is complete. Asynchronous code is NON-BLOCKING, which means that while the request is being made, the rest of the code should carry on normally.

Let’s try this first-hand. In our chooselocation.dart file, lets create a function called as getData().

This function will simulate the act of fetching data from a database. We’ll not actually fetch data, but we will introduce a delay so that it *seems* like the system is busy fetching data. In order to introduce a delay we use a class in Dart called ‘Future’, and it’s method called ‘delayed’. This method accepts 2 parameters.

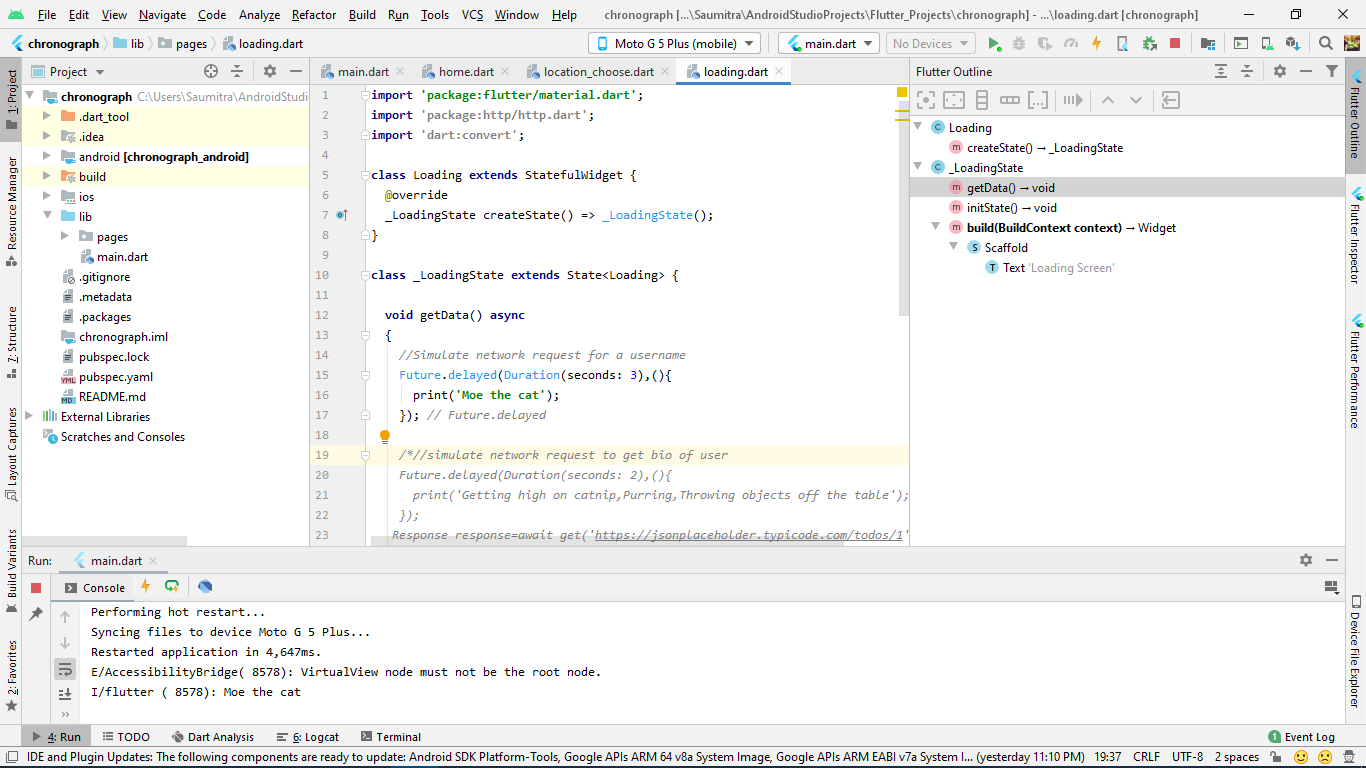
1. The duration—For which we’ll use a Duration object, and specify time in seconds.
2. The function which will execute once the delay is over. (Here we’ll just print some text on the console)

**void** getData(){  
 *//Simulate network request for a username* Future.delayed(Duration(seconds: 3),(){  
 print(**'Moe the cat'**);  
 });  
}

We’ll call this function in the initState function of our class.

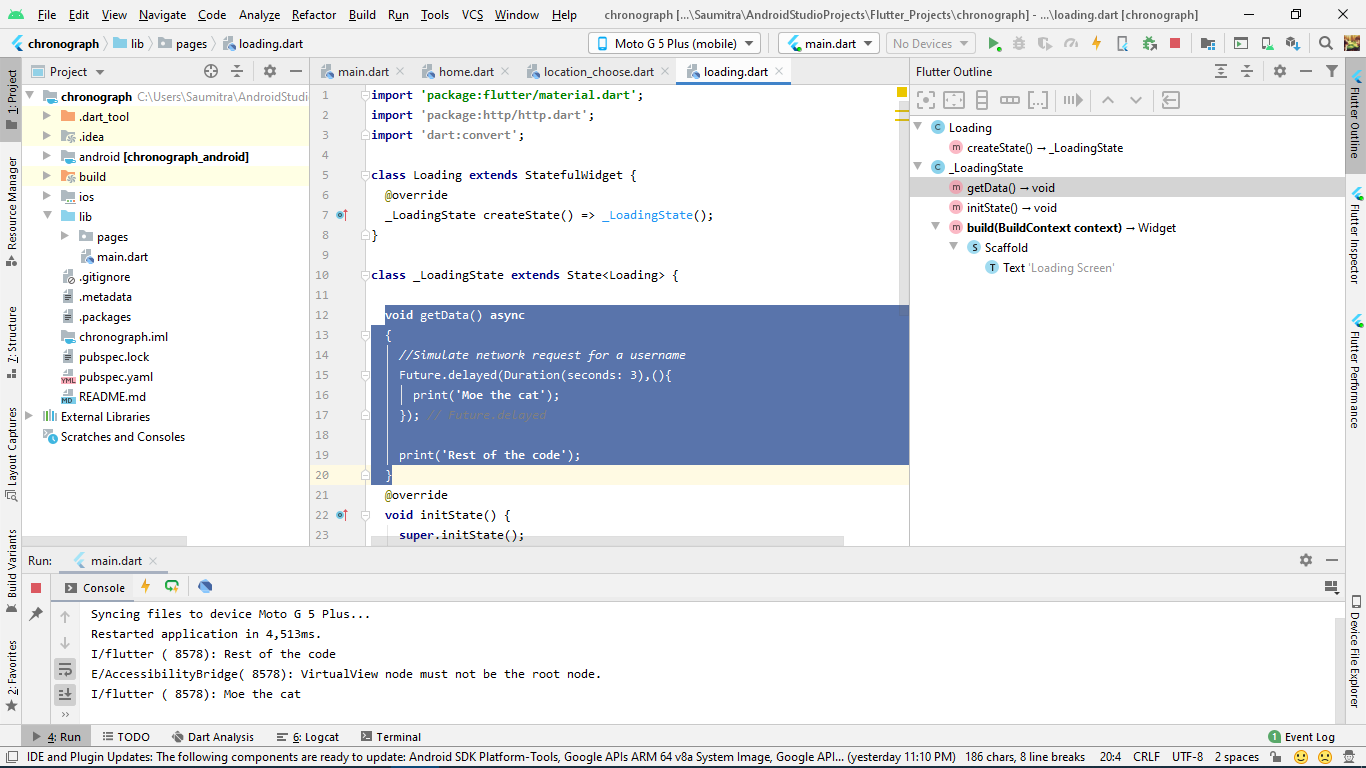
@override  
**void** initState() {  
 **super**.initState();  
 getData();  
}

Our app displays the words ‘Moe the cat’ after 3 seconds.



The cool thing about this is that this is NON-BLOCKING code. So, if we try to print something after that, it will get executed. The app is not waiting for the request to be completed. The rest of the code carries on while the app is fetching the request. For this reason, we shall see the text ‘Rest of the code’ first, and then 3 seconds later, the text ‘Moe the cat’:

**void** getData(){  
 *//Simulate network request for a username* Future.delayed(Duration(seconds: 3),(){  
 print(**'Moe the cat'**);  
 });  
 print(**'Rest of the code'**);  
}



But sometimes, we DO need the next task to wait for the completion of the first one. Suppose the first task fetches the username, and we need this fetched username of the first task to be used in a second task. In this case, the second task cannot execute before the completion of the first task. Thus, the second task is locked in place by the first. In order to do this, we need to specify that our function is an asynchronous function by using the keyword ‘async’.

**void** getData() **async**{

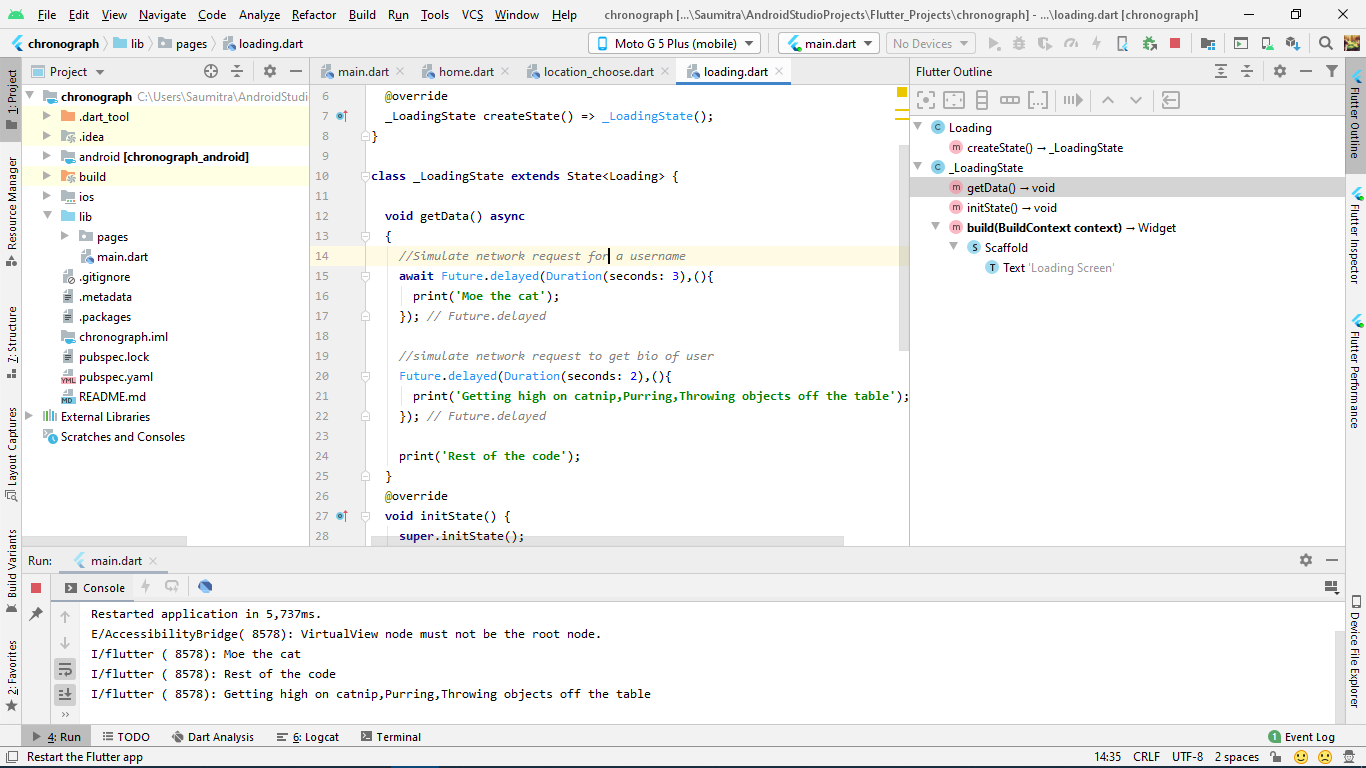
//code inside

}

We also need to specify which statement we need to wait for to complete before the rest of the code runs. This is done by the ‘await’ keyword.

Let’s simulate this by modifying our getData() function:

**void** getData() **async**{  
 *//Simulate network request for a username* **await** Future.delayed(Duration(seconds: 3),(){  
 print(**'Moe the cat'**);  
 });  
  
 *//simulate network request to get bio of user* Future.delayed(Duration(seconds: 2),(){  
 print(**'Getting high on catnip,Purring,Throwing objects off the table'**);  
 });  
  
 print(**'Rest of the code'**);  
}

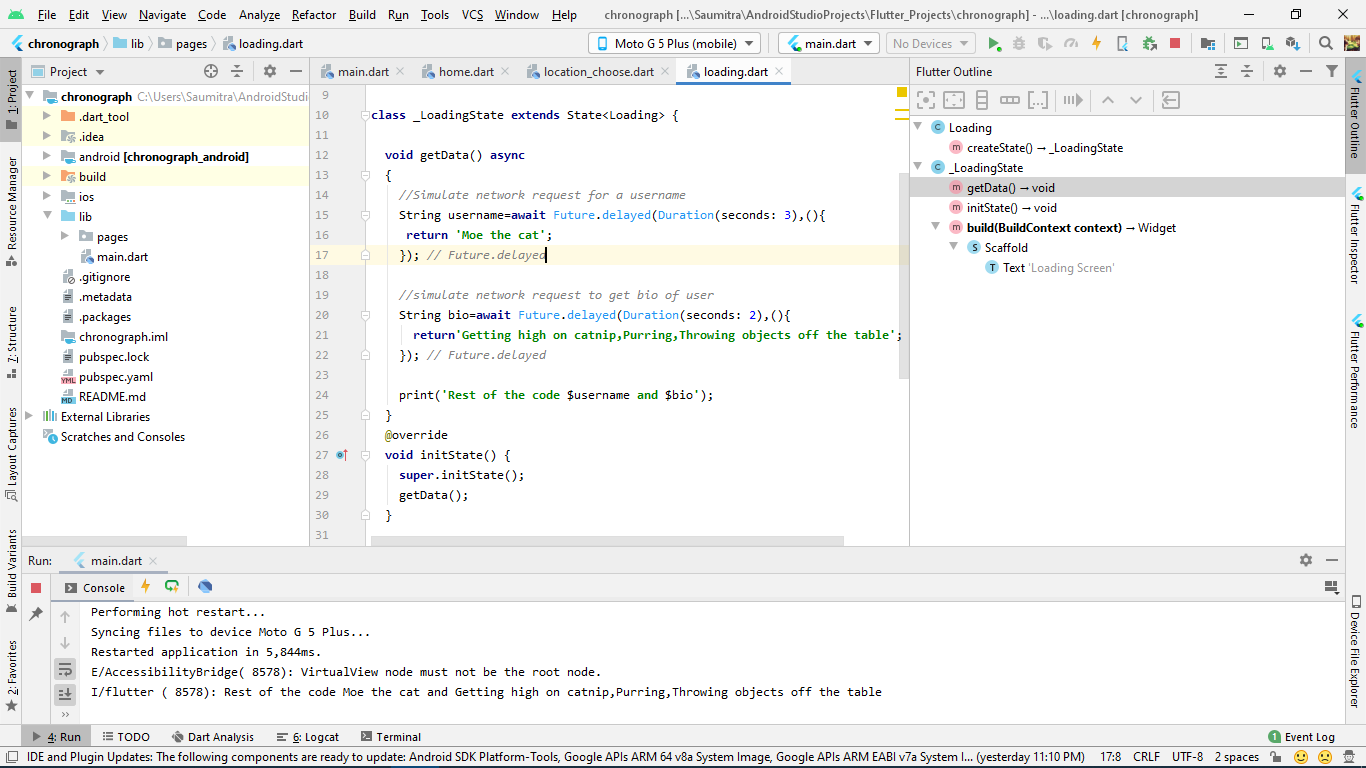


We can see that the app needs to wait for the first task to complete. Then it prints ‘Moe the cat’. After the 1st task is complete, the rest code block executes. Because the 2nd task is non-blocking and needs 2 seconds to complete, ‘Rest of the code’ is displayed first. (The 2 seconds start after the 1st task completes). Then finally after 2 seconds, the output of the 2nd task is displayed.

We could also make these Future.delayed() functions return some values. For example, after the 1st task is complete it returns the username as a string. Similarly, when the second task completes, it will return the bio as a string. We can make both of these tasks as ‘await’ so that they execute one after the other. After both tasks are complete, we can print the username and bio in the last print statement. Our function looks like:

**void** getData() **async**{  
 *//Simulate network request for a username* String username=**await** Future.delayed(Duration(seconds: 3),(){  
 **return 'Moe the cat'**;  
 });  
  
 *//simulate network request to get bio of user* String bio=**await** Future.delayed(Duration(seconds: 2),(){  
 **return'Getting high on catnip,Purring,Throwing objects off the table'**;  
 });  
  
 print(**'Rest of the code** $username **and** $bio**'**);  
}

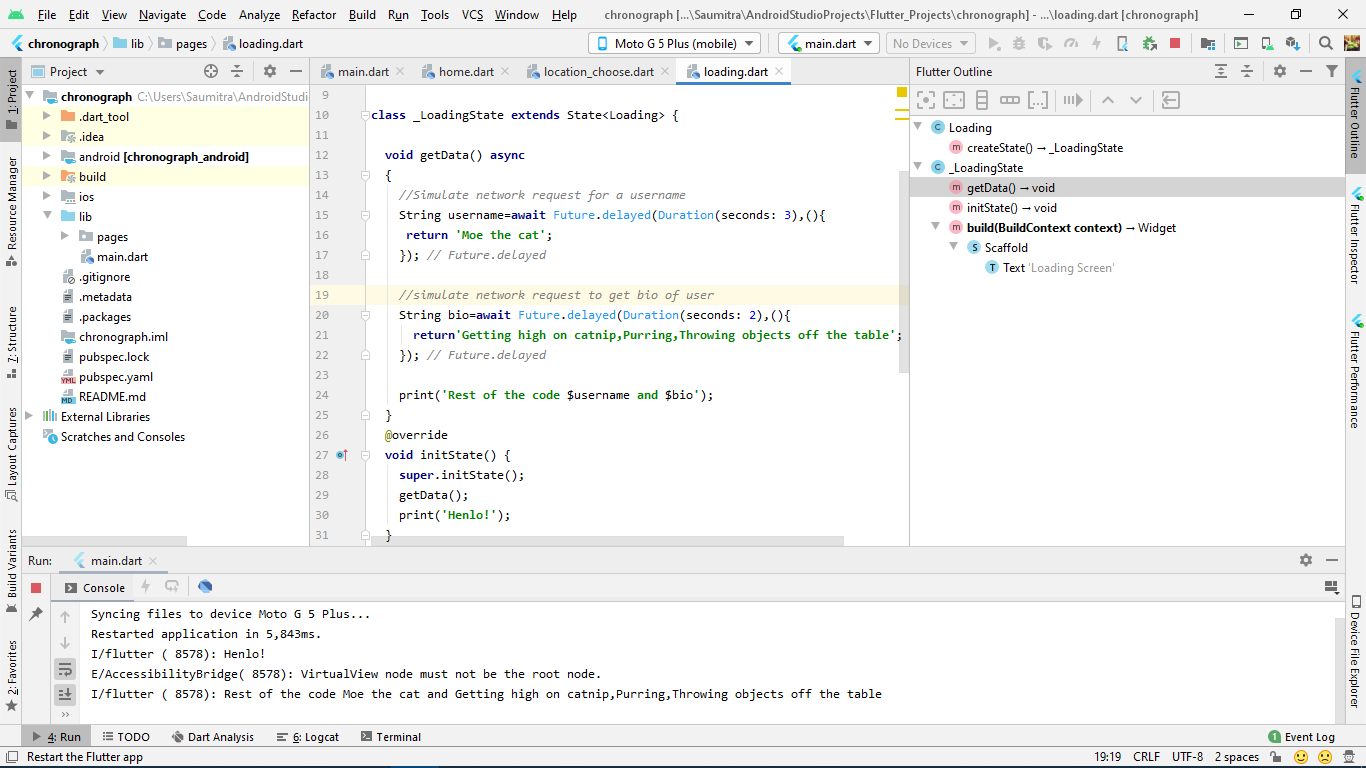
This code will now return the username after 3 seconds, then the password after 2 seconds. After the whole 5 seconds are over, they are displayed in a single string:



The cool thing about this is that, because we have set our getData() function as ‘async’, it will not block any other function in the initState where it is called. Consider an example:

@override  
**void** initState() {  
 **super**.initState();  
 getData();  
 print(**'Henlo!'**);  
}

Async function, therefore will not block the print statement



Flutter Packages:

Flutter packages are bits of code and logic that other developers have already written. This is exactly similar to the dependencies that we used to do in Java-Android. They are external API’s which allow us to perform complex tasks using already written pieces of code like a sliding menu, a date picker, a file uploader, or anything of that sort.

All these packages are released on the package release channel for the dart language called [pub.dev](https://pub.dev/).

In our app, we shall be using a package called ‘http’ which facilitates easy handling of http requests and responses which can be found [here](https://pub.dev/packages/http).

We just add the dependency in our pubspec.yaml file:

**dependencies**:  
 **flutter**:  
 **sdk**: flutter  
 **http**: ^0.12.1

In order to test out our http package, we’ll fetch some test data in the form of a JSON file from an external API. We’ll make our request in the getData function we created earlier, and call the getData function in the initState function. As a result, all our data is fetched before the state object is created.

Just for testing purposes, we’ll fetch data from an API called [JSON Placeholder](https://jsonplaceholder.typicode.com/). It is a fake rest API for testing code. It will just send some dummy JSON file as a response to a request.

First we’ll import our http package like so:

**import 'package:http/http.dart'**;

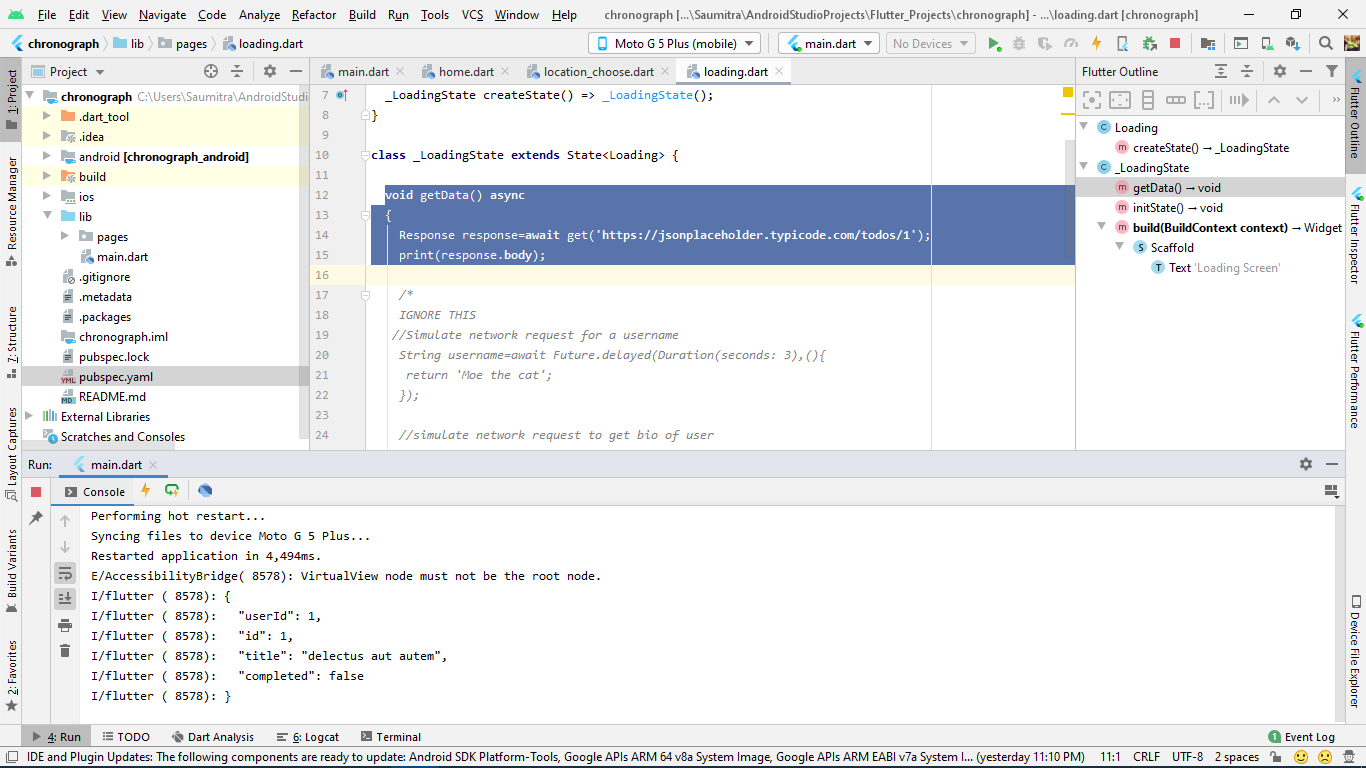
Then we’ll head on to our getData function:

In order to provide a request to an http server, we need to use the get() function from the http package. The output will be stored in an instance of an object of the Response class. We also need to add the await keyword so that all future statements wait for this request to complete. After getting the response, we shall print the body of the response object as it contains the data:

**void** getData() **async**{  
 Response response=**await** get(**'https://jsonplaceholder.typicode.com/todos/1'**);  
 print(response.**body**);

}

We get this as the output on the console:



Currently however, whatever we are getting as the output on the screen is a ‘string’ and not actual JSON. So, we cannot access any of the individual elements of the JSON file. In order to have access, we need to *parse* this JSON file. This JSON file is parsed into a data structure that we can have access to in DART, and which looks similar to a JSON notation. So, which data structure comes to mind?

Of course! It’s a MAP!

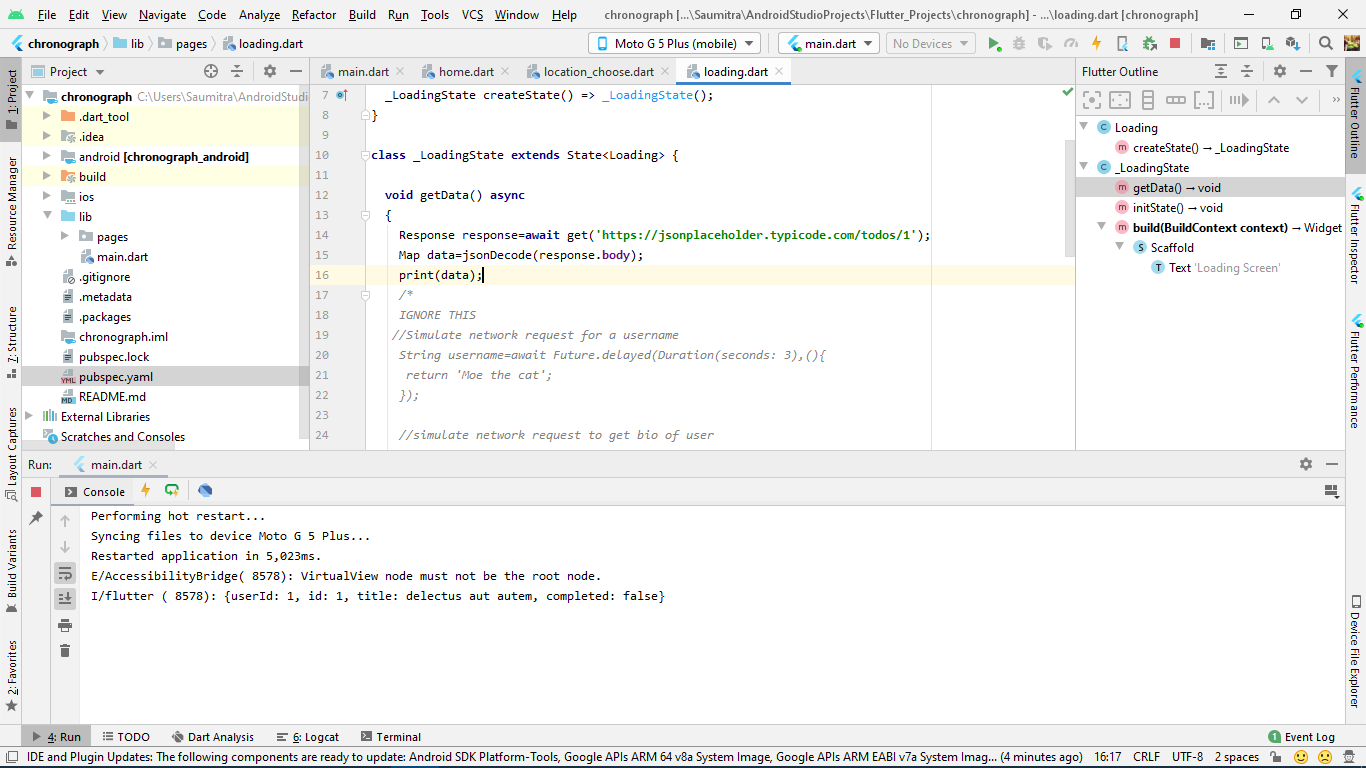
This parsing can be done with the help of a DART method called ‘JSONdecode’. This is included in the DART package called ‘dart:convert’. So we’ll need to import that too.

**import 'dart:convert'**;

The code looks like:

**void** getData() **async**{  
 Response response=**await** get(**'https://jsonplaceholder.typicode.com/todos/1'**);  
 Map data=jsonDecode(response.**body**);  
 print(data);

}



Now that the JSON is converted to a Map, we can access its individual properties:

**void** getData() **async**{  
 Response response=**await** get(**'https://jsonplaceholder.typicode.com/todos/1'**);  
 Map data=jsonDecode(response.**body**);  
 print(data);  
 print(data[**'title'**]);

}

