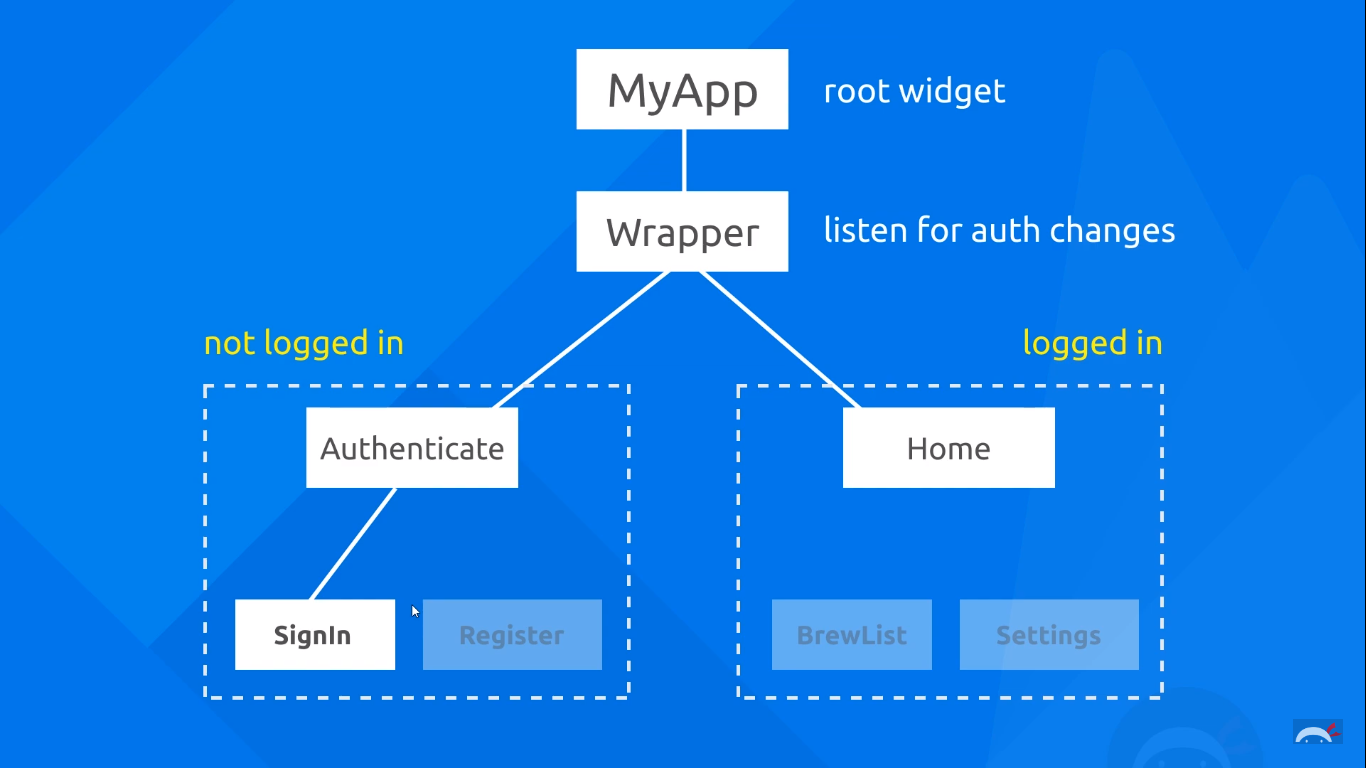
Streams:

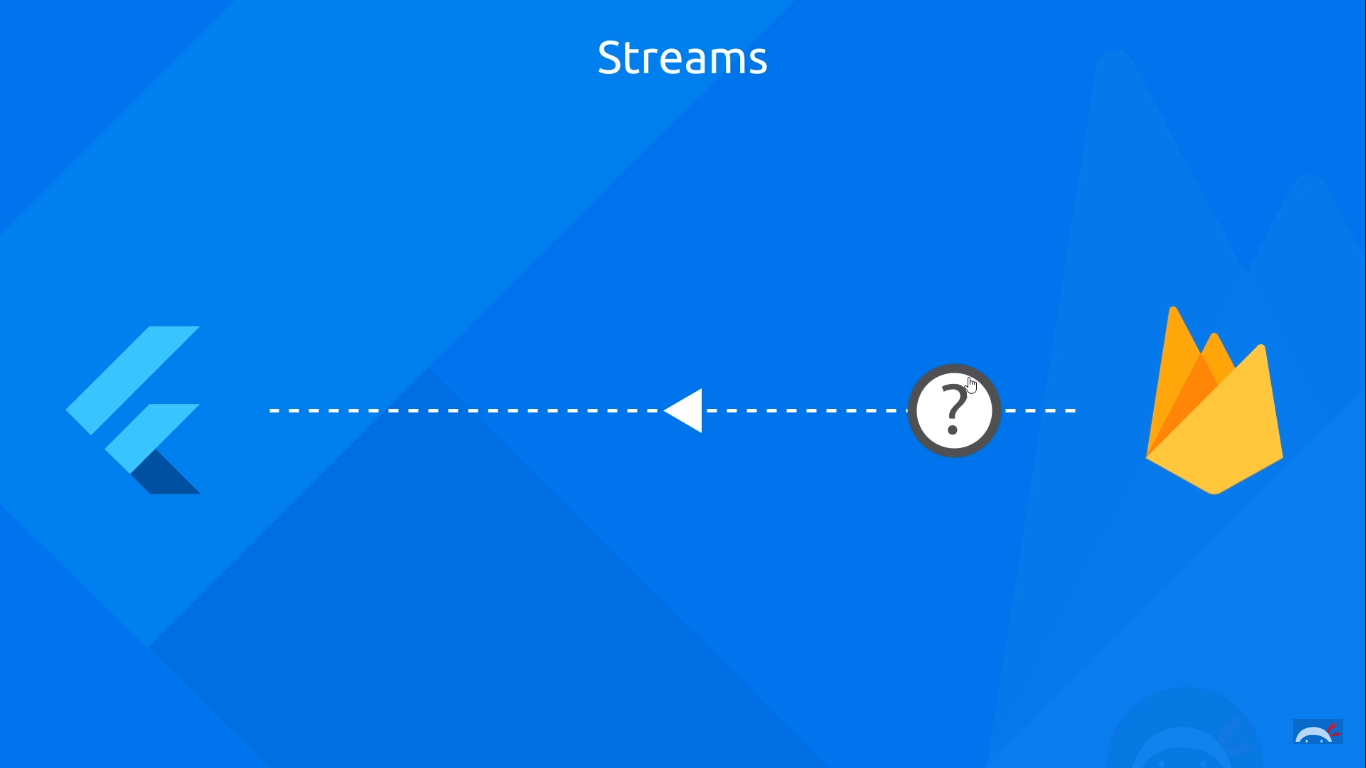
Till now we have developed the app till this stage. We can now sign in anonymously, and in return we get access to the user information in the form of a FirebaseUser object which we then converted into a custom ‘User’ object.



Now what we want to happen is to identify this authentication change, and then display the home screen to the user when he logs in. Moreover, if in the future, we could also listen to THAT authentication change, and display the login screen again to the user. Thus, essentially we are protecting our home screen from unauthenticated uses.

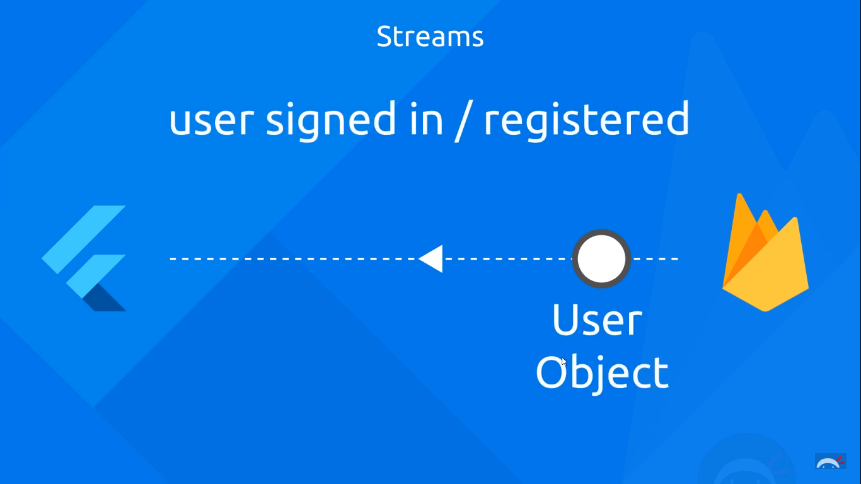
Remember, that in our ‘authenticate.dart’ file, we are supposed to be listening to these authentication changes. We want the app to switch between the home screen and login screen dynamically. In order to do so, we’ll be using a ‘stream’.

Remember what we discussed about Futures? (If not, get a recap [here](https://youtu.be/OTS-ap9_aXc).) We said that a Future was nothing but a box that the calling function receives. When the asynchronous task completes, it adds processed data to this box, and then opens it. The future then returns this data to the caller. Similarly, if the async task ends with an error, the future returns an error to the calling function.



Streams are nothing but Futures, but unlike futures which can only process single data, a ‘stream’ processes a stream of data. This means that it can execute async functions and tasks on data ***AS IT ARRIVES ON THE EVENT LOOP.*** Our auth changes are nothing but streams. It is just piece of data which is generated as the user logs in or out, and we need processing logic to deal with it AS IT HAPPENS—ergo **STREAMS!** A stream has a listen method by which we can listen to the stream data which is returned by a function call (like a file access on a disk). A stream has 2 ends: The ‘Stream’ end – where data arrives, and there is the ‘Sink’ end – where data is inserted.

Learn more in depth about streams [here](https://youtu.be/nQBpOIHE4eE).

The FirebaseAuth service already has a stream that we can listen to. We can do this by invoking the onAuthChanged function. We shall do this in our auth.dart file which has the ‘AuthService’ class, and also has all the methods for signing-in, signing-out,etc.

Now, in order to create a stream we shall use the get keyword to access the Stream end. (Here we are not creating a stream from scratch, but we are essentially hijacking a stream). The syntax for that is:

Stream <Return\_Type> get Stream\_Name {

//code;

//return statement;

}

We shall therefore invoke the onAuthStateChanged() function in order to get the stream. We shall then return this stream. Our stream therefore looks like:

Stream<FirebaseUser> get user{

    return \_auth.onAuthStateChanged;

}

Now, this is absolutely fine. But as we said earlier, we don’t want to work with the FirebaseUser object, but instead, we want to work with our custom user object which just contains the uid as of now. Thus, what we could do is to *map* this stream of FirebaseUser objects to objects of our custom ‘User’ class. We take the help of the ‘map’ function for that.

The map method takes in a function as the parameter, returns the result of this function to its caller.

Stream<FirebaseUser> get user{

    return \_auth.onAuthStateChanged.map((){});

}

Our function will take a FirebaseUser instance as the input parameter, and will return a ‘User’ object as the output. And we already have a function which does that, don’t we? Yes! It’s the userFromFirebaseUser(FirebaseUser user) function.

One more thing, upon doing so, this stream will no longer be a FirebaseUser stream, but instead it will be a ‘User’ stream.

//auth changed user stream

  Stream<User> get user{

    return \_auth.onAuthStateChanged.map((FirebaseUser user)

    {

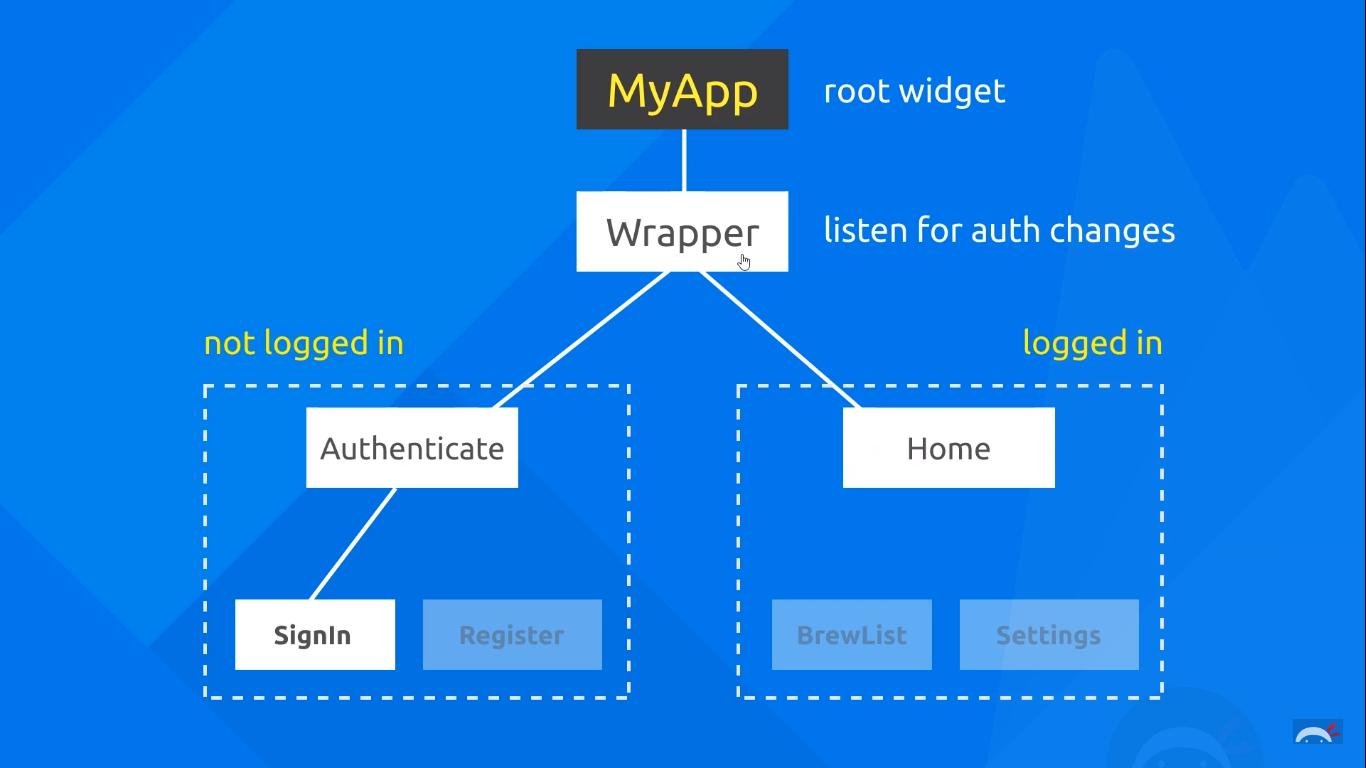
      return \_userFromFirebaseUser(user);

    });

}

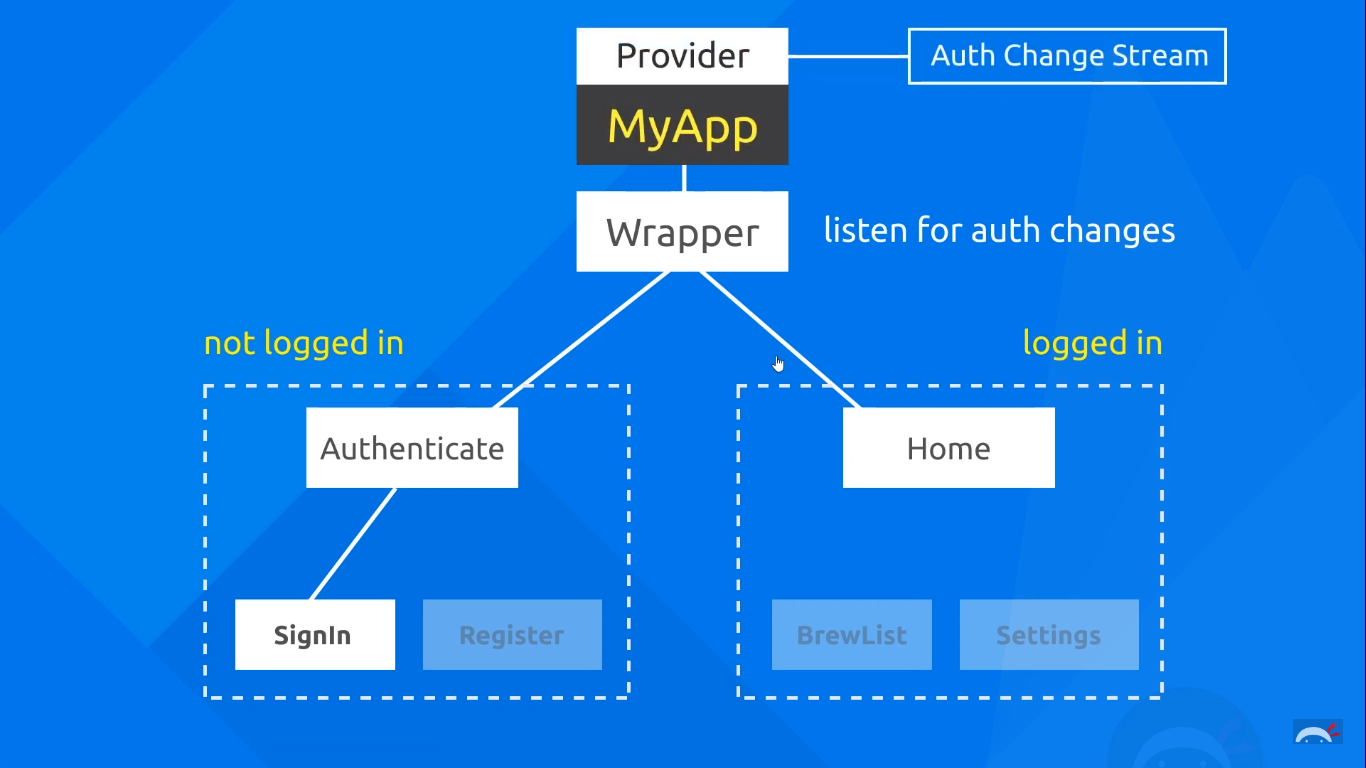
The provider package:

Now we have our ‘auth’ stream set up in our authService class. We just have to find a way to use it in our wrapper class.



As discussed earlier we need the wrapper widget to continuously listen to auth changes. If the object returned by the stream is a ‘null’ value, it implies that the user has logged out, and we need to display the Authenticate() widget. On the other hand, if the authStream returns a valid User object, then we need to display the Home() widget which represents the home screen.

We need a way to access the stream and pass the stream data downwards to all children of the Widget tree. The way we do exactly this is by a package called ‘Provider’.



The ‘provider’ package is a Google recommended solution for state management in flutter. We wrap the root widget with a provider, and pass a stream to it. Whenever we receive data inside the stream, the provider package passes it down the widget tree. Get the provider package [here](https://pub.dev/packages/provider).

Now we head on to the main.dart file and wrap the root widget (i.e. MyApp) with the provider widget. In particular we’ll be using a ‘StreamProvider’ we shall be using a specific method called ‘StreamProvider.value’. This takes in a property called ‘valu’ wherein we specify the stream. In order to specify the stream, we create an anonymous instance of the AuthService class, and access the ‘user’ stream that we just created. We also need to specify what kind of data we’ll be listening to from the stream. This will be data of the type ‘User’. Thus, we’ll specify this to the StreamProvider within angular brackets. The code looks like:

import 'package:cupped\_lightning/models/user.dart';

import 'package:cupped\_lightning/screens/wrapper.dart';

import 'package:cupped\_lightning/services/auth.dart';

import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

void main() {

  runApp(MyApp());

}

class MyApp extends StatelessWidget {

  // This widget is the root of your application.

  @override

  Widget build(BuildContext context) {

//StreamProvider listening to ‘user’ stream of datatype ‘User’

    return StreamProvider<User>.value(

        value: AuthService().user,

          child: MaterialApp(

        home: Wrapper(),

        theme: ThemeData(

    // Define the default brightness and colors.

    brightness: Brightness.dark,

    primaryColor: Colors.pink,

    accentColor: Colors.pinkAccent,

    textSelectionHandleColor: Colors.pinkAccent

    ),

  )

);

  }

}

In order to access the provider data inside the wrapper.dart file, we need to user the Provider.of() method. We’ll invoke this inside the build method as we need the context object.

final user=Provider.of<User>(context);

Since we need to dynamically decide between the Home() and Authenticate widget, we place in a few ‘if’ statements, and our wrapper.dart file looks like:

import 'package:cupped\_lightning/models/user.dart';

import 'package:cupped\_lightning/screens/authenticate/authenticate.dart';

import 'package:cupped\_lightning/screens/home/home.dart';

import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

class Wrapper extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    final user=Provider.of<User>(context);

    print(user);

    //Return either Home or Authenticate widgets

    if(user==null)

    {

      return Authenticate();

    }

    else {

       return Home();

    }

  }

}

Signing Out:

Let’s now add a sign out button to our home screen! We’ll add our sign out button to the AppBar using the actions property which takes in a list of widgets. Currently we only have one button in the list. Flushing out the UI code, our home.dart looks like this:

import 'package:cupped\_lightning/services/auth.dart';

import 'package:flutter/material.dart';

class Home extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      backgroundColor: Colors.grey[900],

      appBar: AppBar(

      title: Text('Cupped Lightning'),

      backgroundColor: Colors.grey[850],

      elevation: 0.0,

      actions: <Widget>[

        FlatButton.icon(

          onPressed: () {},

        icon: Icon(

          Icons.person,

          color: Colors.grey[200]

        ),

         label: Text(

           'Log Out',

           style: TextStyle(

             color: Colors.grey[200]

           ),

           ),

         )

      ],

      ),

    );

  }

}

Now, inside the AuthService class in auth.dart we’ll need to define the signOut method. Here we shall call the signOut method of the FirebaseAuth object. This method, unlike the signIn method doesn’t return any value. The sign out method looks like:

Future signOut() async{

    try{

      return await \_auth.signOut();

    }

    catch(e)

    {

      print(e.toString());

      return null;

    }

  }

}

In the home.dart file, we’ll need an instance of the AuthService class in order to access this signOut method. So, that’s what we’ll do:

AuthService \_auth=AuthService();

We’ll also need to make an asynchronous call to this signOut method in the onPressed property of the signout button. It looks like:

FlatButton.icon(

          onPressed: () async{

           await \_auth.signOut();

        },

        icon: Icon(

          Icons.person,

          color: Colors.grey[200]

        ),

         label: Text(

           'Log Out',

           style: TextStyle(

             color: Colors.grey[200]

           ),

           ),

         )

Our entire home.dart file looks like:

import 'package:cupped\_lightning/services/auth.dart';

import 'package:flutter/material.dart';

class Home extends StatelessWidget {

  AuthService \_auth=AuthService();

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      backgroundColor: Colors.grey[900],

      appBar: AppBar(

      title: Text('Cupped Lightning'),

      backgroundColor: Colors.grey[850],

      elevation: 0.0,

      actions: <Widget>[

        FlatButton.icon(

          onPressed: () async{

           await \_auth.signOut();

        },

        icon: Icon(

          Icons.person,

          color: Colors.grey[200]

        ),

         label: Text(

           'Log Out',

           style: TextStyle(

             color: Colors.grey[200]

           ),

           ),

         )

      ],

      ),

    );

  }

}



Sign-In and Register Forms:

Currently we have completed the anonymous sign-in and sign-out. What we want to do, is to have a username-password combination for the user to sign-in and register. For that we’ll have to create the respective forms. So, in the signIn.dart file, we’ll update our UI to include a Form() widget. Inside the form widget, we can use multiple widgets like the textInput box, the submit button etc. Currently we are inserting a column widget into the child property of the form as we want the username password text fields to be displayed one below the other.

The textinput in flutter is a widget called TextFormField(). Inside this, we need to specify what happens when the text input in the form field changes. Thus, every time the user types in a key or deletes the text in the text box, the corresponding function is called. This property is called the onChanged property. It takes in a function which has the value of the form field as the parameter and inside that function, we’ll do something with the entered text.

In the password TextFormField, we can set the property of obscureText to true in order to hide the input from the screen, so that the password looks like dots. Our code looks like:

import 'package:cupped\_lightning/services/auth.dart';

import 'package:flutter/material.dart';

class SignIn extends StatefulWidget {

  @override

  \_SignInState createState() => \_SignInState();

}

class \_SignInState extends State<SignIn> {

  @override

  Widget build(BuildContext context) {

   final AuthService \_authService= AuthService();

    return Scaffold(

      backgroundColor: Colors.grey[900],

      appBar: AppBar(

        backgroundColor: Colors.grey[850],

        elevation: 0.0,

        title: Text('Sign In'),

      ),

      body: Container(

        padding: EdgeInsets.symmetric(vertical:20.0 ,horizontal:50.0) ,

        child: Form(

          child: Column(

            children: <Widget>[

              SizedBox(height: 20.0),

              TextFormField(

                onChanged: (value){},

              ),

              SizedBox(height: 20.0),

              TextFormField(

                obscureText: true,

                onChanged: (value){},

              ),

              SizedBox(height: 20.0),

              RaisedButton(

                onPressed: (){},

                color: Colors.pink,

                child: Text('Sign In'),

                ),

            ],

            )

          )

        /\*

Commenting out the earlier anonymous sign in button

RaisedButton(

          onPressed: () async{

            dynamic result =await \_authService.signInAnonymous();

            if(result==null)

            {

              print("Error Signing In");

            }

            else

            {

              print("Sign In Successful");

              //print(result.uid);

            }

          },

          child: Text('Sign In Anonymously'),

        ),

        \*/

      ),

    );

  }

}

Now, we need to record the username and password entered by the user somewhere. So, we’ll create two string variables to store the information as the user types in.

//text field state

   String email=' ';

   String password=' ';

Now, we update the onChanged properties of the two TextFormFields by setting their data into these two strings.

child: Form(

          child: Column(

            children: <Widget>[

              SizedBox(height: 20.0),

              TextFormField(

                onChanged: (value){

                  setState(() {

                    email=value;

                  });

                },

              ),

              SizedBox(height: 20.0),

              TextFormField(

                obscureText: true,

                onChanged: (value){

                  setState(() {

                    password=value;

                  });

                },

              ),

An interesting thing to note here is that we have used the setState() function, even if we aren’t changing any of the UI widgets. setState triggers the build method to rebuild the widget, or in this case, the value of that widget. Technically you can't actually make changes to widgets. If we don’t perform a setState(), widgets don't get updated in the right manner. Thus, to avoid possible errors in the setting of the username & password values, we use setState() as a safety measure.

As a test, we’ll print out the username and password values entered on the console.

RaisedButton(

                onPressed: ()async

                {

                  print(email);

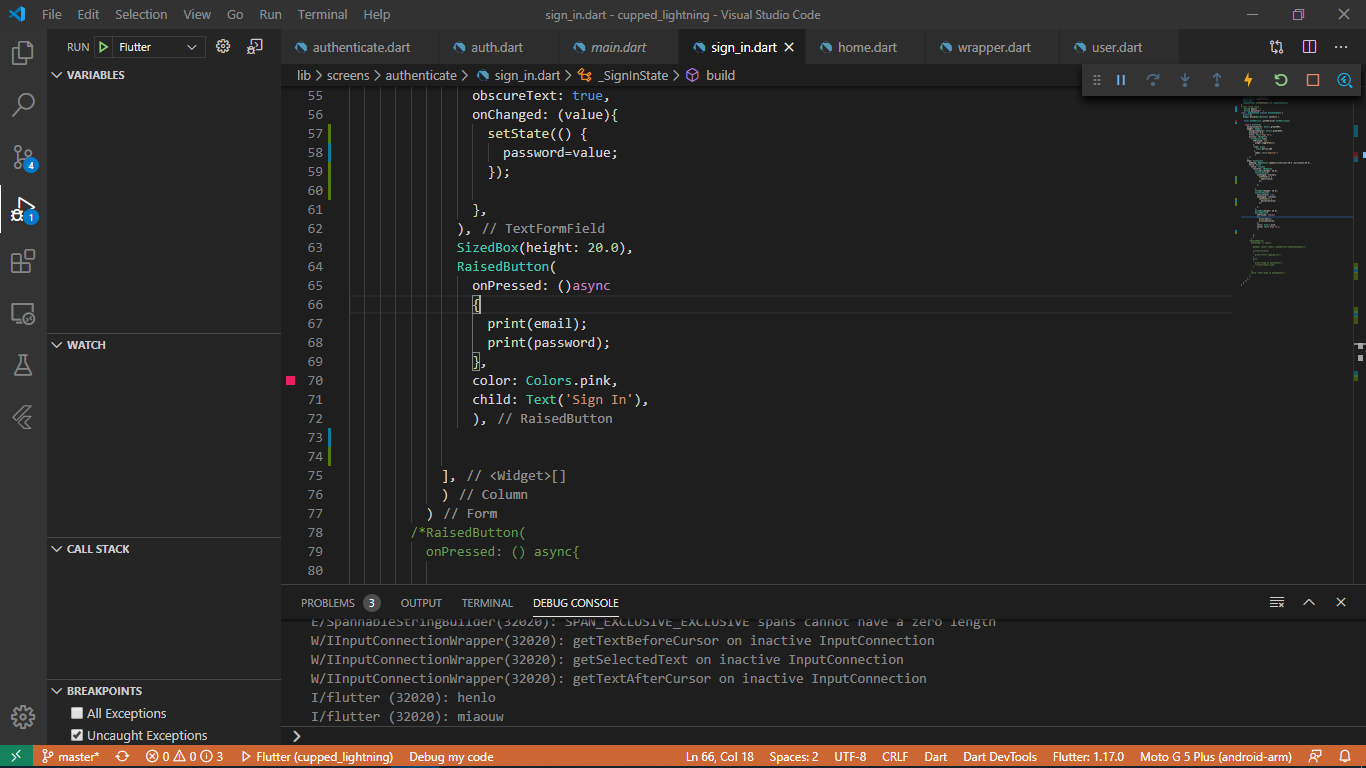
                  print(password);

                },

                color: Colors.pink,

                child: Text('Sign In'),

                ),



Thus, our sign in page looks like this:

import 'package:cupped\_lightning/services/auth.dart';

import 'package:flutter/material.dart';

class SignIn extends StatefulWidget {

  @override

  \_SignInState createState() => \_SignInState();

}

//text field state

   String email=' ';

   String password=' ';

class \_SignInState extends State<SignIn> {

  @override

  Widget build(BuildContext context) {

   final AuthService \_authService= AuthService();

    return Scaffold(

      backgroundColor: Colors.grey[900],

      appBar: AppBar(

        backgroundColor: Colors.grey[850],

        elevation: 0.0,

        title: Text('Sign In'),

      ),

      body: Container(

        padding: EdgeInsets.symmetric(vertical:20.0 ,horizontal:50.0) ,

        child: Form(

          child: Column(

            children: <Widget>[

              SizedBox(height: 20.0),

              TextFormField(

                onChanged: (value){

                  setState(() {

                    email=value;

                  });

                },

              ),

              SizedBox(height: 20.0),

              TextFormField(

                obscureText: true,

                onChanged: (value){

                  setState(() {

                    password=value;

                  });

                },

              ),

              SizedBox(height: 20.0),

              RaisedButton(

                onPressed: ()async

                {

                  print(email);

                  print(password);

                },

                color: Colors.pink,

                child: Text('Sign In'),

                ),

            ],

            )

      ),

    );

  }

}

In the exact same manner, we’ll create a register page with the same code. This will be saved in a new dart file called register.dart inside the screens/authenticate folder.

import 'package:cupped\_lightning/services/auth.dart';

import 'package:flutter/material.dart';

class Register extends StatefulWidget {

  @override

  \_RegisterState createState() => \_RegisterState();

}

//text field state

   String email=' ';

   String password=' ';

class \_RegisterState extends State<Register> {

   final AuthService \_authService= AuthService();

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      backgroundColor: Colors.grey[900],

      appBar: AppBar(

        backgroundColor: Colors.grey[850],

        elevation: 0.0,

        title: Text('Sign Up'),

      ),

      body: Container(

        padding: EdgeInsets.symmetric(vertical:20.0 ,horizontal:50.0) ,

        child: Form(

          child: Column(

            children: <Widget>[

              SizedBox(height: 20.0),

              TextFormField(

                onChanged: (value){

                  setState(() {

                    email=value;

                  });

                },

              ),

              SizedBox(height: 20.0),

              TextFormField(

                obscureText: true,

                onChanged: (value){

                  setState(() {

                    password=value;

                  });

                },

              ),

              SizedBox(height: 20.0),

              RaisedButton(

                onPressed: ()async

                {

                  print(email);

                  print(password);

                },

                color: Colors.pink,

                child: Text('Register'),

                ),

            ],

            )

          )

      ),

    );

  }

}

Toggling between forms:

Now that we have our signIn and register screens set up, we need a way to toggle between these screens in a fluid manner. This means that if a user already has an account, he could easily click a button on the register screen to sign in using his credentials. Similarly if a user doesn’t have an account, he’ll click on the register icon to create a new account. We’ll set up a toggling method in the authenticate.dart file which is the parent widget of both these screens.

We’ll first add these buttons onto the appbar. These are called appBar actions and are included inside an ‘actions’ property on the appBar. The signIn() widget will have a register button, and the register() widget will have a sign-in button.

//Sign In Page

appBar: AppBar(

        backgroundColor: Colors.grey[850],

        elevation: 0.0,

        title: Text('Sign In'),

        actions: <Widget>[

          FlatButton.icon(

            onPressed: (){

            },

             icon: Icon(

               Icons.person\_add

             ),

              label: Text('Register')

              )

        ],

//Register Page

appBar: AppBar(

        backgroundColor: Colors.grey[850],

        elevation: 0.0,

        title: Text('Sign Up'),

        actions: <Widget>[

          FlatButton.icon(

            onPressed: (){

              widget.toggleFunc();

            },

             icon: Icon(

               Icons.person

             ),

              label: Text('Sign In')

              )

        ],

      ),

Now, in order to toggle between these screens we need to set up a check in the authenticate.dart screen. We’ll create a Boolean value called showSignIn() and initially set it to true. We’ll Then we’ll include a simple if-else check to show the screens.

bool showSignIn=true;

if(showSignIn)

    {

      return SignIn();

    }

    else

    {

      return Register();

    }

We’ll also need to create a function which will toggle this showSignIn Boolean. It will essentially negate the value of the showSignIn variable. This function will be triggered whenever the user taps on either the signIn or register appBar actions.

void toggleView()

  {

    setState(() {

      showSignIn=!showSignIn;

    });

  }

Now in order to call this function from the signIn() and register() widgets, we’ll need access to it. In order to do so, we’ll pass this function as a parameter (or a property) to these widgets, like so:

import 'package:cupped\_lightning/screens/authenticate/register.dart';

import 'package:cupped\_lightning/screens/authenticate/sign\_in.dart';

import 'package:flutter/material.dart';

class Authenticate extends StatefulWidget {

  @override

  \_AuthenticateState createState() => \_AuthenticateState();

}

class \_AuthenticateState extends State<Authenticate> {

  bool showSignIn=true;

  void toggleView()

  {

    setState(() {

      showSignIn=!showSignIn;

    });

  }

  @override

  Widget build(BuildContext context) {

    if(showSignIn)

    {

      return SignIn(toggleFunc: toggleView);

    }

    else

    {

      return Register(toggleFunc: toggleView);

    }

  }

}

Since we’re passing it as a property, we’ll need to define the respective variable and receive it in the constructor. So that’s what we’ll do in both of our widgets. Remember that we need to define this function object in the WIDGET class and not in the state-object class:

SIGN-IN SCREEN:

import 'package:cupped\_lightning/services/auth.dart';

import 'package:flutter/material.dart';

class SignIn extends StatefulWidget {

  final Function toggleFunc;

Define the variables and constructor here

  SignIn({this.toggleFunc});

  @override

  \_SignInState createState() => \_SignInState();

}

//text field state

   String email=' ';

   String password=' ';

class \_SignInState extends State<SignIn> {

  @override

  Widget build(BuildContext context) {

NOT HERE

   final AuthService \_authService= AuthService();

    return Scaffold(

      backgroundColor: Colors.grey[900],

. . . rest of the code

REGISTER SCREEN:

mport 'package:cupped\_lightning/services/auth.dart';

import 'package:flutter/material.dart';

class Register extends StatefulWidget {

  final Function toggleFunc;

  Register({this.toggleFunc});

  @override

  \_RegisterState createState() => \_RegisterState();

}

//text field state

   String email=' ';

   String password=' ';

class \_RegisterState extends State<Register> {

   final AuthService \_authService= AuthService();

  @override

  Widget build(BuildContext context) {

    return Scaffold(

. . . rest of the code

Now, in order to call this function after pressing the button on the appBar, we’ll need to write it as widget.toggleFunc().

SIGN IN SCREEN:

import 'package:cupped\_lightning/services/auth.dart';

import 'package:flutter/material.dart';

class SignIn extends StatefulWidget {

  final Function toggleFunc;

  SignIn({this.toggleFunc});

  @override

  \_SignInState createState() => \_SignInState();

}

//text field state

   String email=' ';

   String password=' ';

class \_SignInState extends State<SignIn> {

  @override

  Widget build(BuildContext context) {

   final AuthService \_authService= AuthService();

    return Scaffold(

      backgroundColor: Colors.grey[900],

      appBar: AppBar(

        backgroundColor: Colors.grey[850],

        elevation: 0.0,

        title: Text('Sign In'),

        actions: <Widget>[

          FlatButton.icon(

            onPressed: (){

              widget.toggleFunc();

            },

             icon: Icon(

               Icons.person\_add

             ),

              label: Text('Register')

              )

        ],

      ),

      body: Container(

        padding: EdgeInsets.symmetric(vertical:20.0 ,horizontal:50.0) ,

        child: Form(

          child: Column(

            children: <Widget>[

              SizedBox(height: 20.0),

              TextFormField(

                onChanged: (value){

                  setState(() {

                    email=value;

                  });

                },

              ),

              SizedBox(height: 20.0),

              TextFormField(

                obscureText: true,

                onChanged: (value){

                  setState(() {

                    password=value;

                  });

                },

              ),

              SizedBox(height: 20.0),

              RaisedButton(

                onPressed: ()async

                {

                  print(email);

                  print(password);

                },

                color: Colors.pink,

                child: Text('Sign In'),

                ),

            ],

            )

          )

      ),

    );

  }

}

REGISTER SCREEN:

import 'package:cupped\_lightning/services/auth.dart';

import 'package:flutter/material.dart';

class Register extends StatefulWidget {

  final Function toggleFunc;

  Register({this.toggleFunc});

  @override

  \_RegisterState createState() => \_RegisterState();

}

//text field state

   String email=' ';

   String password=' ';

class \_RegisterState extends State<Register> {

   final AuthService \_authService= AuthService();

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      backgroundColor: Colors.grey[900],

      appBar: AppBar(

        backgroundColor: Colors.grey[850],

        elevation: 0.0,

        title: Text('Sign Up'),

        actions: <Widget>[

          FlatButton.icon(

            onPressed: (){

              widget.toggleFunc();

            },

             icon: Icon(

               Icons.person

             ),

              label: Text('Sign In')

              )

        ],

      ),

      body: Container(

        padding: EdgeInsets.symmetric(vertical:20.0 ,horizontal:50.0) ,

        child: Form(

          child: Column(

            children: <Widget>[

              SizedBox(height: 20.0),

              TextFormField(

                onChanged: (value){

                  setState(() {

                    email=value;

                  });

                },

              ),

              SizedBox(height: 20.0),

              TextFormField(

                obscureText: true,

                onChanged: (value){

                  setState(() {

                    password=value;

                  });

                },

              ),

              SizedBox(height: 20.0),

              RaisedButton(

                onPressed: ()async

                {

                  print(email);

                  print(password);

                },

                color: Colors.pink,

                child: Text('Register'),

                ),

            ],

            )

          )

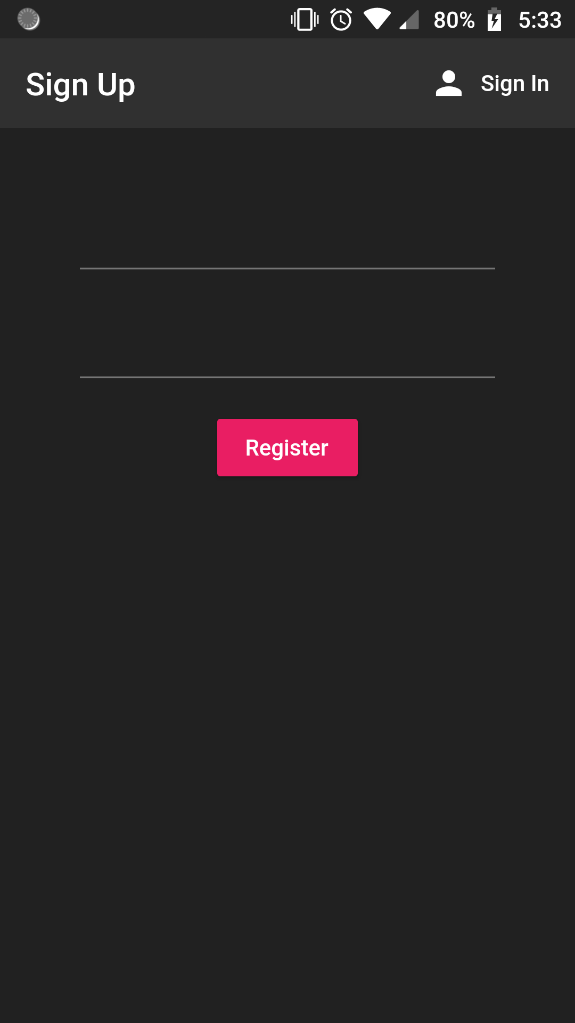
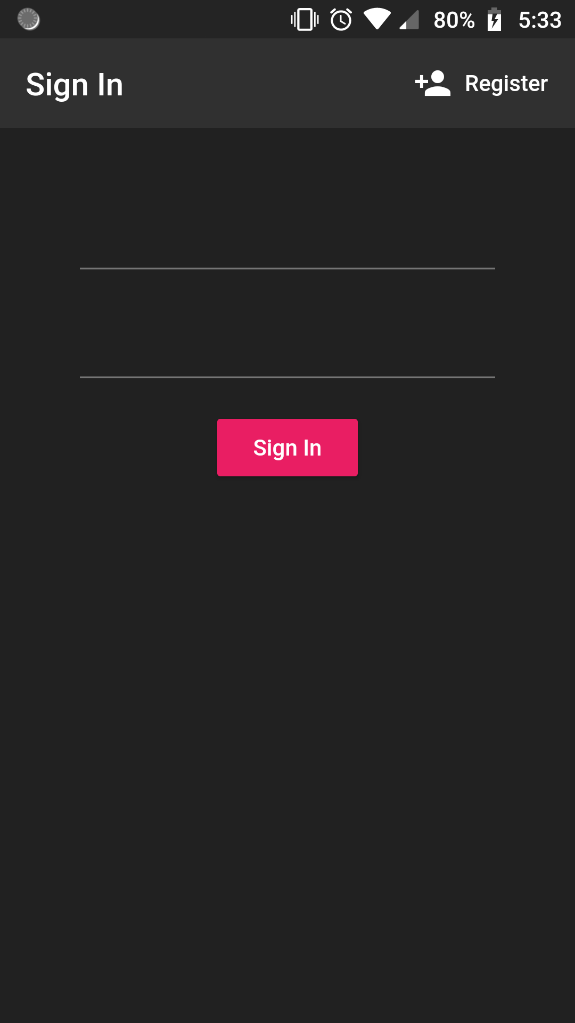
      ),

    );

  }

}

The output looks like:

Register with Email and Password:

Before we proceed with connecting the register screen with Firebase, we need to perform some validations of our own. Here, in our form widget, we shall first setup a form key. This form key will help us keep track of the state of our form. Thus, whenever the user enters information in the form, at every key press, the state of the form changes, and this form key keeps track of it. This is a global key that we have just declared.

final \_formKey=GlobalKey<FormState>();

We also need to specify inside the form widget that we are using this variable called ‘\_formKey’ as the key for our form.

body: Container(

        padding: EdgeInsets.symmetric(vertical:20.0 ,horizontal:50.0) ,

        child: Form(

          key: \_formKey,

. . . Rest of the code

Now, in the register button, we also need to invoke the validate() function before proceeding with the registration. This will locally validate our form fields before the data is sent to the firebase server. Thus, if the form is validated, the code proceeds, otherwise, it returns an error. This can be implemented using a simple if statement. If the textboxes are valid, then the validate function returns a null. Otherwise, it returns an error.

RaisedButton(

                onPressed: ()async

                {

                  if(\_formKey.currentState.validate())

                  {

//Code for proceeding with registration

}

}

color: Colors.pink,

                child: Text('Register'),

             ),

At this point of time, we haven’t specified what exactly we want to validate. We have just set up the form key, and have ensured that the form details will be submitted only when the form is valid.

For every textbox we need to implement a property called ‘validator’. This property takes in a function as the value. This function has the text inside the box as the input parameter. Inside the function we specify what we want to validate. If the validation condition is satisfied, the function returns null. Otherwise, we can return a string as the error. The awesome thing about the TextFormField is that this error string will be displayed below the text-box, like in many of the commercial apps. For example, we want to ensure that the email text box is not empty, we can do it using:

 TextFormField(

                validator: (value){

                  if(value.isEmpty)

                  {

                    return 'Email field cannot be empty';

                  }

                  else

                  {

                    return null;

                  }

                },

                onChanged: (value){

                  setState(() {

                    email=value;

                  });

                },

              ),

Similarly, we can ensure that the password is at least 6 characters long. We can do this by;

TextFormField(

                obscureText: true,

                validator: (value){

                  if(value.length<6)

                  {

                    return 'Password should be at least 6 chars long';

                  }

                  else

                  {

                    return null;

                  }

                },

                onChanged: (value){

                  setState(() {

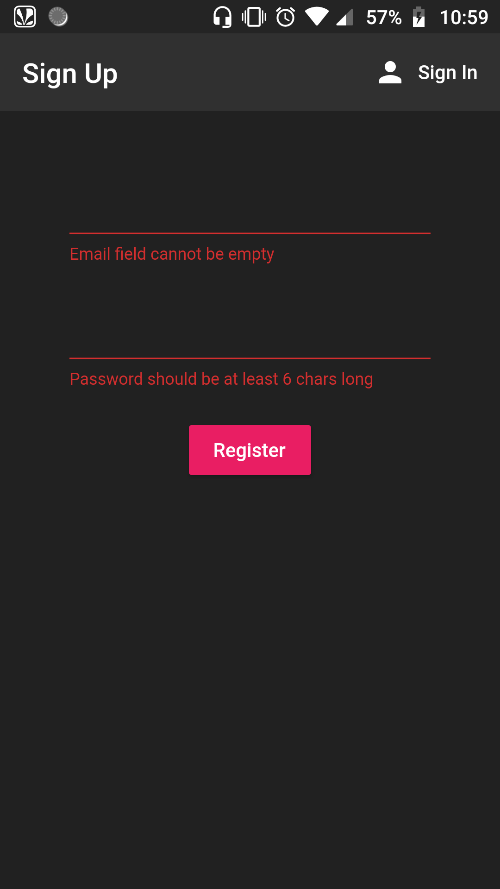
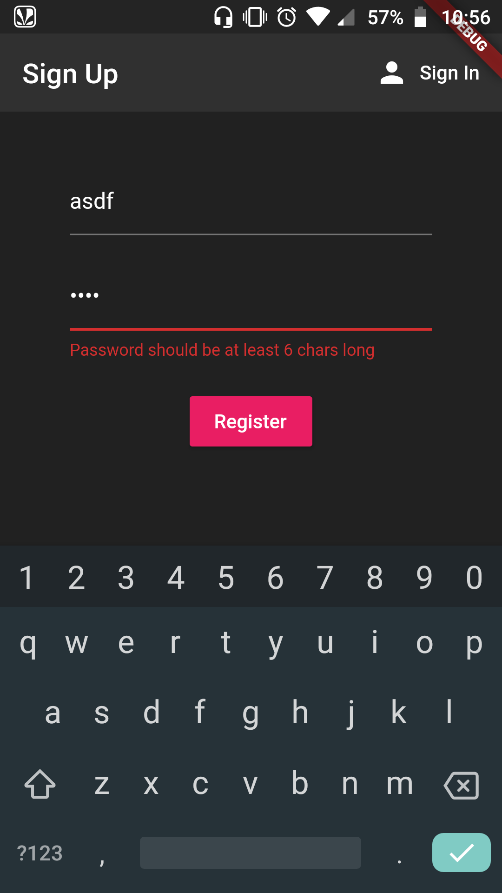
                    password=value;

                  });

                },

              ),

Upon testing our output looks like:

Now we shall define our registration function in our auth.dart file inside the AuthService class. This is very similar to the anonymous sign in, and we should be able to follow along just by studying the code:

//register with email & password

  Future registerWithEmailAndPassword(String email, String password) async {

    try

    {

    AuthResult result=await \_auth.createUserWithEmailAndPassword(email: email, password: password);

    FirebaseUser user=result.user;

    return \_userFromFirebaseUser(user);

    }

    catch(e)

    {

      print(e.toString());

      return null;

    }

  }

Coming back to our register() widget, we need to call this function after pressing the submit button, after the if condition which validates the text inserted in the text fields. We should also place in a text widget which displays an error if the registration fails. For this we define an empty ‘error’ string initially.

String error='';

We also need to insert an if statement which handles the condition when the registration function returns with a null value. In that case we need to return an error.

RaisedButton(

                onPressed: ()async

                {

                  if(\_formKey.currentState.validate())

                  {

                    dynamic result= await \_authService.registerWithEmailAndPassword(email, password);

                    if(result==null)

                    {

                      setState(() {

                        error='Registration failed';

                      });

                    }

                  }

                },

                color: Colors.pink,

                child: Text('Register'),

                ),

                SizedBox(height: 12.0),

                Text(

                  error,

                  style: TextStyle(

                    color: Colors.red,

                    fontSize: 14.0

                  ),

                  )

If we enter an invalid email id, the firebase auth returns an error (essentially a null result) which makes the error string as ‘Registration failed’. This string is then displayed in the text box below:

