Disaster Scenario: Smart Balloons

Team: Smartphone App

In the Things Section, write a list of the hardware, software apps, online services you used for the project. Were there any specific requirements to use them? Use free text if you want to. English and German are both welcome. PM Team will take care of the rest.

Things:

1. Flutter (Dart)
2. Firebase Database
3. Git/GitHub
4. VisualStudio Code/IntelliJ IDEA
5. Photoshop

Describe your activity in the project using steps, that answer the following questions (similar to a tutorial):

1. What was your goal?
2. How did you start working toward your goal?
3. Which technologies/frameworks/example projects did you use?
4. Why did you choose this specific technology/method?
5. What problems did you encounter? How did you solve them?
6. Pro-tip for people who will follow your steps?

Story:

**Intro:** Create an “easy on the eyes” view for the app.

We set on blueish tones as a representation of the sky. We decided we would use cards to show each of the real-time parameters. Setting these parameters isn’t very hard as you just need to change the styles in Dart, similar as you would do in CSS.

We struggled with unfamiliarity with the language overall, but there were plenty of online resources to help and Dart allows you to develop seamlessly for both Android and iOS (and web, to some extent).

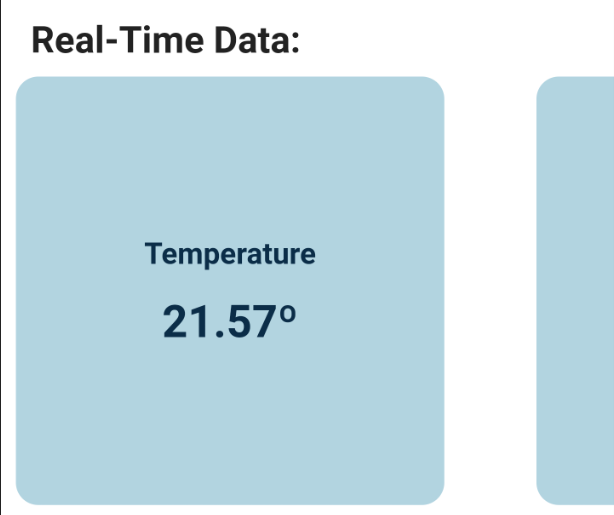
*Example sentences (not obligatory to use):*

* *We built a view that used cards to show real time data with horizontal scrolling.*

**Steps:**

1. Use the **Scaffold Widget** to set the base view of the app – Gives you parameters for an **AppBar** and a **Body**, among others that we did not use, to set the style exactly as you want.
2. Use the **GridView** **Widget** for the horizontal scrolling. The parameter scrollDirection lets you choose if you want a vertical or horizontal scrolling.
3. With a **GestureDetector** you can set up the tapping to change views. In our case, the tapping takes you to a different view with the graphs for that specific parameter. This is done via the *onTap* parameter which takes a function.
4. Each “card” is a **Container Widget** with **BoxDecoration** to set parameters like the colour and radius. Padding and BorderRadius are examples of parameters we set to achieve the look you can see below.
5. Every card is inside of a **FutureBuilder**, because each of these cards needs to get data from the database and this widget allows us to display something different while the data is being fetched. In our case, it’s a circular progress indicator, similar to those you see in most modern apps, which is done by just calling the *CircularProgressIndicator()* function from the material package.

**Result:**



Story:

**Intro:**

Create an easy way to understand where the clusters physically are. It also creates an easy way to choose which cluster to display the information from.

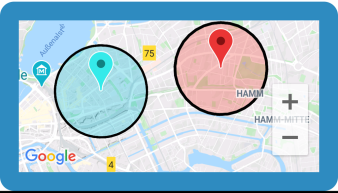
*Example sentences (not obligatory to use):*

* *We built a map that used circles* and *markers to display* the geographical position of the clusters.

**Steps**

1. Inside the **Body** we create a **Container Widget** for the **Map**.
2. Use the **google maps(google\_maps\_flutter: ^1.0.6)** plugin to display the **map** as an **GoogleMap Widget**.
3. Use the plugin **integrated** functionsto draw **Circles** and pinpoint **Markers** on the map to represent the clusters.
4. Because of some initialization issues we use a **condition** to substitute the **map** with a **message** informing the user that the map is being loaded

**Result**



Story:

**Intro**: Connect the app to the Firebase Database.

Grab the latest values of each parameter to display on the cards. To connect to the database, we used the **firebase\_database** package available on the official Flutter/Dart package website (<pub.dev>). We would recommend the use of the **FutureBuilder Widget**, to avoid issues with the asynchronous nature of the database like we explained in one of the previous stories.

Since Firebase Database is non-relational, it is formatted as a JSON file, which was in theory, easier to work with, and since Firebase and Flutter are both Google products, we figured it would be easier to work with both at the same time.

**Steps:**

1. On this [Firebase Documentation](https://firebase.google.com/docs/flutter/setup?platform=android) page, you can follow the tutorial to connect to your database, which is what we did. You just need to:
   1. Register your app with the database by giving it your package name or application ID in your project home in Firebase;
   2. Add a Firebase config file to your project in the android/app directory of your Flutter app.
2. Make a reference to the spot in the database to access. We did this by having a variable store our reference to the database (*Firebase.instance.reference()*) and then accessing the specific point we wanted with the *.child(‘nameInFirebase’)* function. If you only want the latest value, you can limit it by using the *.limitToLast(1)* method, which is what we did.
3. Grab the values from the database by using the snapshot you get. You can access your value (or values if multiple) by mapping them to dynamic datatypes like this: *Map<dynamic, dynamic> values = snapshot.data.value;*
   1. There are different ways to access the database depending on what data you want or how the data is organized in the database. This was, at the time, the best solution for us. If you only wanted one value, for example, it doesn’t make sense to assign it to a map.

**Result:**

You have the data from the database and can do with it as you please. Having access to the database and getting permanent values is sometimes difficult, so to overcome this we had to use something like *FirebaseAnimatedList()* which let us correct a bug where data wasn’t actually being stored. By creating the reference on the *initState()* we also corrected some different bugs with the calls to the database.

Story:

**Intro**: Make a new view for the graphs.

Using the **charts\_flutter** package, we were able to plot graphs more easily with data we wanted. We recommend using the **Stateful Widget** as your base instead of the **Stateless** as it is much easier to mutate if/when necessary even though it takes more work in the beginning to get it up and running. It also allows you to free memory at your command, which might be necessary.

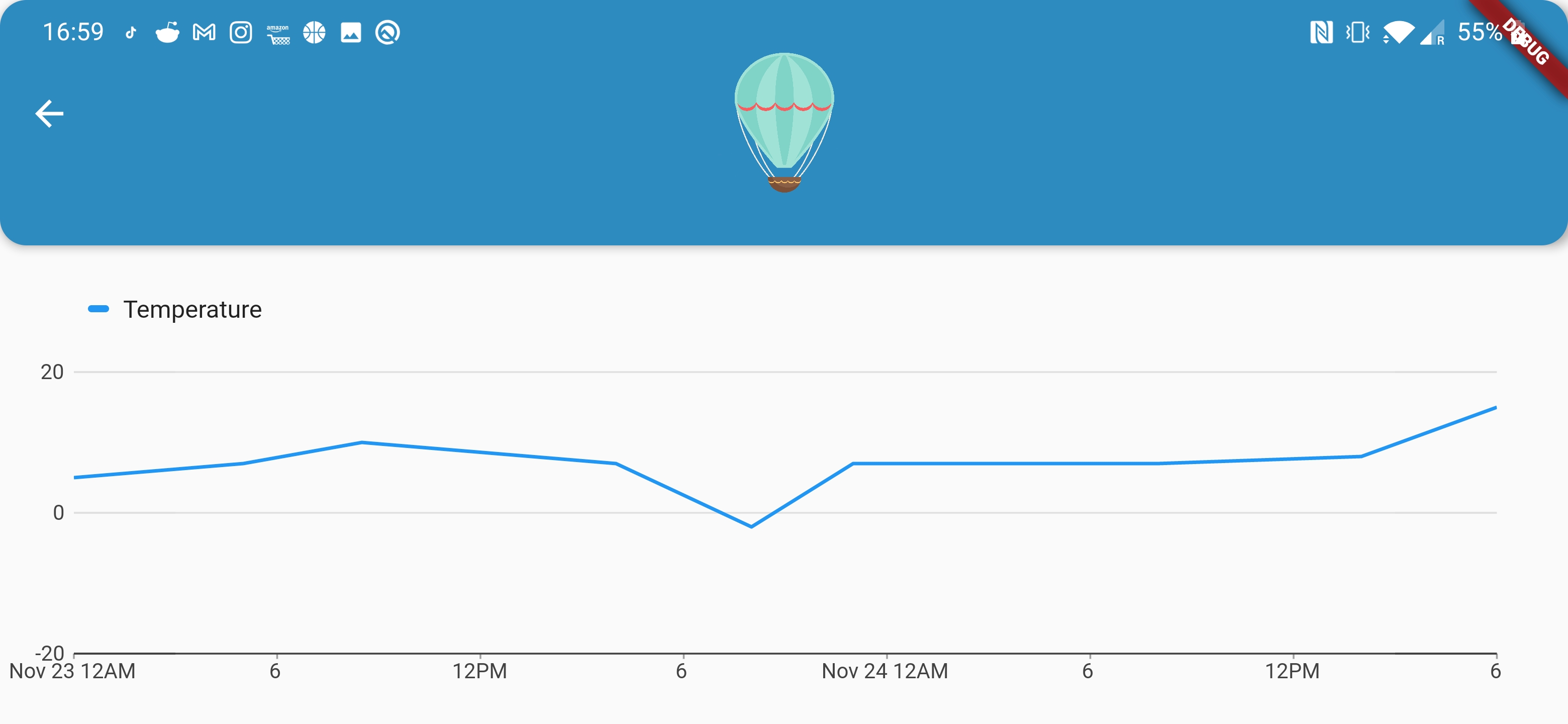
We then used **DateTime** values to get the X axis and the parameter values for the Y axis.

After building your new view you can use the *Navigator.push(…);* to go from the main view to this view and the *Navigator.pop(…);* to go the other way around.

**Steps:**

1. Setup your new view as a **Stateful Widget** and build it with your parameters (we recommend basing it on your main view.
2. Setup your *SeriesList* and calculate any values you need for the XY axis values in helper functions and then set the charts parameters (id, colour, etc.) from the charts\_flutter package on the *charts.TimeSeriesChart()* function.
   1. For us, this meant grabbing all the necessary values of a certain parameter from the database and then set them in different lists.

**Result:**



Story:

**Intro:** Set up device orientation for different views.

We wanted to force portrait orientation on the main view but have the possibility to use the phone in landscape mode when on the graphs view. This meant that we had to change some things. To avoid any problems in the future, we recommend setting up the*initState()*override of your *\_Builder*class in each view (make sure to call *super.initState();* as well). To use the *initState()* you have to have a **Stateful Widget**. In here it’s easy to set it up with the *SystemChrome.setPreferredOrientations* method and use any combination of these orientations:

* *DeviceOrientation.portraitDown*
* *DeviceOrientation.portraitUp*
* *DeviceOrientation.landscapeLeft*
* *DeviceOrientation.landscapeRight*

This way, you can force any of the orientations you want. For us, we used only the portrait up option on the main view, but the 4 on the graphs view. With this you can even force a view on portrait and a different one on landscape, for example, and are not constrained to any of them for your entire app.

**Steps:**

1. Set your *initState()* override if you haven’t yet
2. Inside your *initState()*, set your preferred orientations based on the possibilities above
3. Don’t forget to call for your *dispose()* override and put your device orientation back to the way it was if it needs to!

**Result:**

You will be able to set orientations as you please.