CYNERGY CODING CLUB

Flutter: Session 1

Contents

Introduction (Minimal)

- Flutter is Google’s UI toolkit for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase.
- Flutter has support for converting already existing apps to flutter easily.
- Dart is a programming language for flutter because it is optimized for client development.
- Flutter has hot reload[1] which makes everything simpler.
- Dart compiles to native machine code for mobile, desktop and JS for the web.
- They are trying to make it so we can use flutter to create applications everywhere.

Keep it Simple, State

State is one of the core concepts that make flutter easy to learn. In essence, state is the information that can be read synchronously when the widget is built and might change during the lifetime of the widget. In other words, a state of a widget holds the widget’s properties. In order to change a property of a widget, the state of that widget has to be changed. Given this definition, two cases can arise. The first, the state is fixed once it is declared, and the second, the state can change over time.

Stateful Widget

A stateful widget is a widget whose state can change whenever the developer desires it. These widgets have functional components or other widgets present that do some change to the states of the (stateful) widgets. The state of a stateful widget is represented by its object variables. Changing the value of those object variables means changing the state of the widget.

For example, consider this scenario. There is a button whose text toggles between “ON” and “OFF” whenever it is tapped. The text of the button is a property of the button. Therefore, given the previous definition of state, changing the text means changing the state of the button.

A stateful widget is defined in Dart as follows:

```dart
class MyStatefulWidget extends StatefulWidget {
  @override
  _MyStatefulWidgetState createState() => _MyStatefulWidgetState();
}

class _MyStatefulWidgetState extends State<MyStatefulWidget> {
  @override
  Widget build(BuildContext context) {
    return Container(

    );
  }
}
```

As seen above, there are two things that need to be considered. One, the MyStatefulWidget class, and the other, _MyStatefulWidgetState class. Due to the presence of these two classes, one can infer that the stateful widget and the state of the stateful widget are separate. This implies that a single stateful widget can have multiple state classes that define its state. But for simplicity, its best to stick to one state class defining the state of one stateful widget.

The MyStatefulWidget class extends the StatefulWidget class, in other words, MyStatefulWidget now has the code necessary to implement a stateful widget. If there are any parameters that need to be passed to the widget to initialize it, the parameters live in this class (More of this will be explained in the upcoming sessions). The only method one needs to care about for now is the createState() method. This method tells flutter which State class should be used as a state for this widget. In the given code, the createState() function returns an object of _MyStatefulWidgetState() class.

The _MyStatefulWidgetState class extends State (The State<MyStatefulWidget> syntax can be thought of as "A state of MyStatefulWidget"). This method will contain the code to build the user interface of this stateful widget (typically consists of a combination of flutter's inbuilt widgets) and contains variables that define the state of the widget (in case of the button example mentioned previously, variables that contain the text of the button to be displayed and so on). In this class, there are three methods that one has to take care of. One, the initState() function, two build() function, and three, dispose() function.

The initState() function is called when the state is being initialized. This method is typically used to initialize any object variables you might have. If there are no object variables, then this method need not be written. If implemented, this function **always** needs a super.initState() function call to call the initState() in State class. The initState() method is defined as follows:

```
@override
void initState() {
  super.initState(); // compulsory
  <statements>
}
```

The build() function is called after initState() finishes execution. There are other situation when this method is called as well, such as after a setState() call. This method returns an object of type Widget, which will be the user interface. This method has one parameter, a

BuildContext. The build context keeps track of where to insert the Widget returned by the build method in the widget tree [2]. The build() method is defined as follows:

```
@override
Widget build(BuildContext context) {
  <statements>
   return <Widget>;
}
```

The dispose() function is called when the stateful widget is removed from the widget tree permanently. Once dispose() is called, the mounted property of StatefulWidget is set to false. Any custom clean up tasks can be implemented in this function. If there are no such tasks, this method can be skipped. But, if its implemented, this function **always** needs a super.dispose() function call to call the initState() in State class. The dispose() method is defined as:

```
@override
void dispose() {
  super.dispose(); // compulsory
  <statements>
}
```

Another function that is important and will be used often is the setState() function. This function is used when the state of the widget has to be changed, and the change should be reflected in the UI as well. The syntax of the setState() function is as follows:

```
setState(() {
      <statements>
});
```

The setState() function takes a function as a parameter. In this function, the change that has to be done is implemented. Taking the example of the button mentioned previously, the setState() call would look something like this:

```
setState(() {
      if (text == "ON")
        text = "OFF";
      else
        text = "ON";
});
```

The `setState()` function will execute the function which is passed in the parameter, and will eventually call the `build()` method to build the widget (and its children, if any) again, so that the changes will be reflected on the UI as well.

it should be noted that inefficient use of the `setState()` call can degrade performance, especially for a large widget tree. This is because the `build()` method is called every time `setState()` is used, and if the widget tree is large, the amount of time taken to rebuild the widget will be significant. If this happens, one solution would be to split the stateful widget into smaller stateful widgets, thereby decreasing the size of the widget tree that needs to be rebuilt.

In summary:

- Stateful widget is a widget whose state can change
- Implemented in two parts, The `StatefulWidget` class, and the State class
- State class encapsulates the state of the widget
- Three functions that are called during the life cycle of the widget:
    - `initState()`: To Initialize the state
    - `build()`: To build the widget UI
    - `dispose()`: Called when the widget is being removed and its life cycle ends.
- `setState()` is called when the state of the widget has to be changed and the change is to be reflected on the UI.
- Inefficient use of `setState()` can degrade performance.

Stateless Widget

A Stateless widget is the opposite of a stateful widget. The state of a stateless widget cannot be changed once it is created. In the sense, if the button mentioned previously was a stateless widget, the text of the button cannot change.

Due to this, stateless widgets do not have an `initState()` method or a `dispose()` method. The `setState()` call will not work in stateless widgets. They do however, have a `build()` method, whose job remains the same, to build the UI.

So, in summary:

- Stateless widgets are widgets whose state is fixed.
- Stateless widgets do not have an `initState()`, `dispose()` or `setState()` methods. They do however, have a `build()` method which needs to be implemented.

Widgets covered

The widgets covered in this session were:

`MaterialApp` widget

Makes an app which adheres to material design concept (The design concept for all android apps). It adds the material design specific functionality (such as the basic animations) to the widgets you program. The widget is defined as:

```
Widget app = MaterialApp(
  home: <widgets>
);
```

The `home` attribute of this widget is the widget you wish to put in the app.

`Scaffold` widget

Material apps follow the material design. The key features of a material design application are: A uniform theme for the whole app, an app bar, a body for the app, a floating action button, and a bottom navigation bar. All this usually requires a lot of boiler plate code. The `Scaffold` widget includes the code for all these functionalities. Therefore, you don’t need to code them yourselves. The widget is defined as:

```
Widget home = Scaffold(
  appBar: <appBar>,
  body: <body>,
  floatingActionButton: <floatingActionButton>
);
```

The `appBar` attribute defines the app bar that will be used. The `body` attribute defines the body of the app, and the `floatingActionButton` attribute defines a floating action button, if any such button is required.

`AppBar` widget

An app bar that follows the material design. An `AppBar` widget has three common attributes, `leading`, `title` and `actions`. These three attributes can be any widget. The `AppBar` is defined as:

```
Widget bar = AppBar(
  leading: <widget>,
  title: <title>,
  actions: <Widget>[
    <widgets>
  ],
);
```

The `<Widget> []` syntax basically means a list of type `Widget`. An example of an `AppBar` is as follows:

```
Widget bar = AppBar(
  title: Text("App Bar Test"),
  leading: Padding(
    padding: EdgeInsets.only(left: 12),
    child: IconButton(
      icon: Icon(Icons.sync),
      onPressed: () {
        print('Click leading');
      },
    ),
  ),
  actions: <Widget>[
    IconButton(
      icon: Icon(Icons.search),
      onPressed: () {},
    ),
    IconButton(
      icon: Icon(Icons.star),
      onPressed: () {},
    )
  ],
),
```

The App bar will look like this:

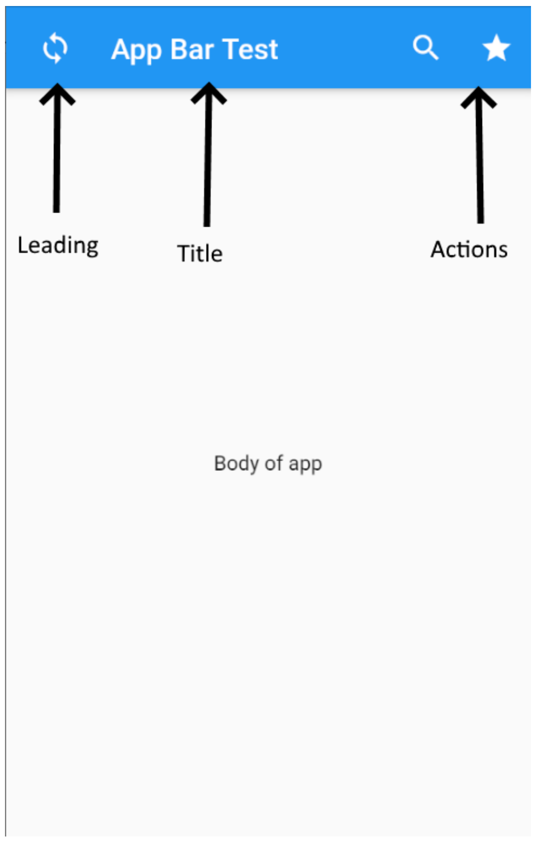

Figure 1: App bar

Brief explanation of the widgets used in the example:

- `Padding`: A widgets that adds padding to its child. The padding is determined by the \`padding\` attribute. The \`padding\` attribute takes an `EdgeInsets` object. This may sound scary, but there are only three things you need to remember.
    - `EdgeInsets.only(right: <value>, left: <value>, top: <value>, bottom: <value>)`

      Specifies padding for each side separately. The <value> can be an integer or a float.
    - `EdgeInsets.symmetric(horizonal: <value>, vertical: <value>)`

      Specifies padding symmetrically. \`horizontal\` attribute applies the same amount of padding to both left and right sides, \`vertical\` attribute applies the same amount of padding to both top and bottom sides.
    - `EdgeInsets.all(<value>)`

      Specifies the same amount of padding for all sides.

- `IconButton`: Basically, means a button with an icon. Takes two attributes, `icon` and `onPressed`.
    - `icon` attribute: specifies what icon should be displayed on the button
    - `onPressed` attribute: specifies what should happen when the button is pressed. It is a function.
- `Icon`: represents an icon.
- `Icons`: Contains some inbuild icons.

`Center` widget

As the name says, places its child at the centre. The widget is defined as:

```
Widget bar = Center(
  child: <widget>
)
```

`Column` widget

Places its children in a column. The widget is defined as:

```
Widget col = Column(
  mainAxisAlignment: <MainAxisAlignment>,
  children: <Widget>[
      <widgets>
  ],
);
```

The `mainAxisAlignment` attribute specifies how the children should be placed along the main axis. Its value can be `MainAxisAlignment.center, MainAxisAlignment.start, MainAxisAlignment.end, MainAxisAlignment.spaceAround, MainAxisAlignment.spaceEvenly, MainAxisAlignment.spaceBetween.`

The main and cross axis can be represented as follows:

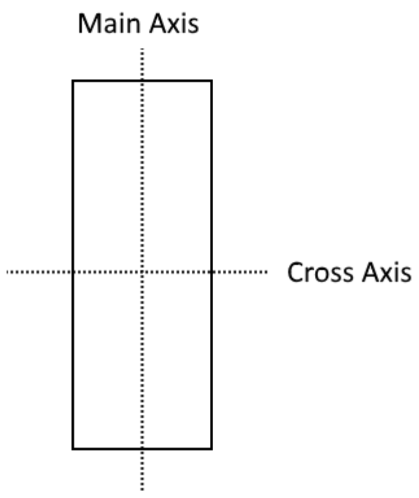

Figure 2: Main and cross axis for column

Row Widget

Places its children in a row. The widget is defined as:

```
Widget row = Row(
  mainAxisAlignment: <MainAxisAlignment>,
  children: <Widget>[
      <widgets>
  ],
);
```

The `mainAxisAlignment` attribute specifies how the children should be placed along the main axis. Its value can be `MainAxisAlignment.center, MainAxisAlignment.start, MainAxisAlignment.end, MainAxisAlignment.spaceAround, MainAxisAlignment.spaceEvenly, MainAxisAlignment.spaceBetween.`

The main and cross axis can be represented as follows:

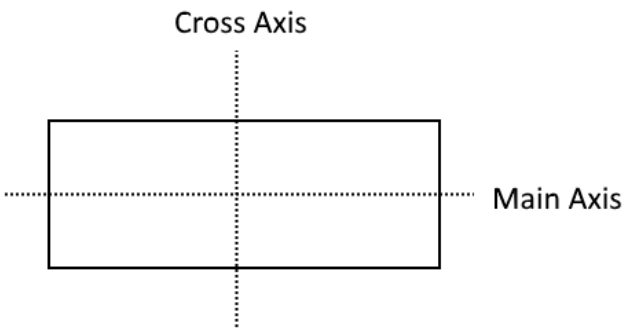

Figure 3: Main and cross axis for row

Container Widget

A widget used to contain other widgets. It is a convenience widget that combines common painting, positioning and sizing widgets. Containers with no children try to be big as possible, unless the incoming size constraints are unbounded, in which case they try to be as small as possible. The widget is defined as:

```
Widget container = Container(
  child: <widget>,
);
```

There are other attributes that Container has. For additional information, refer to the documentation.

Text Widget

Defines a text that is to be displayed. Takes in other attributes such as `TextStyle`, which is used to add style to the text, such as the font name, font size, font style (bold, italics, underline), etc. The widget is defined as:

```
Widget text = Text(
  "some text",
  style: TextStyle(
    <define style attributes>
  ),
);
```

Task

This week’s task is to finish the tic-tac-toe app.

Requirements

Functional Requirements:

- The app should have a tic-tac-toe grid where the user can tap and play his turn.
- The app should alternate between players after every tap.
- The app should display a win message when a player wins the game.

Non-functional requirements:

- The app should be appealing to the eyes (in other words, it should look nice)

Hints and Tips

- One-third of the code is already done during the session and is present in the repo.
- Take a look at `GestureDetector` widget and `InkWell` widget.
- Take a look at `Image` widget as well.

Appendix

Hot Reload

Flutter's hot reload feature helps you quickly and easily experiment, build UIs, add features, and fix bugs. Hot reload works by injecting updated source code files into the running Dart Virtual Machine (VM). After the VM updates classes with the new versions of fields and functions, the Flutter framework automatically rebuilds the widget tree, allowing you to quickly view the effects of your changes. In other words, the moment you save (or hot reload manually by pressing the 'r' key), the changes will instantly take place on the emulator (or phone).

Widget Tree

Widget tree is a structure that represents how our widgets are organized. This is similar to DOMs (Document Object Model) for webpages.

The widget tree for the program in the repo is:

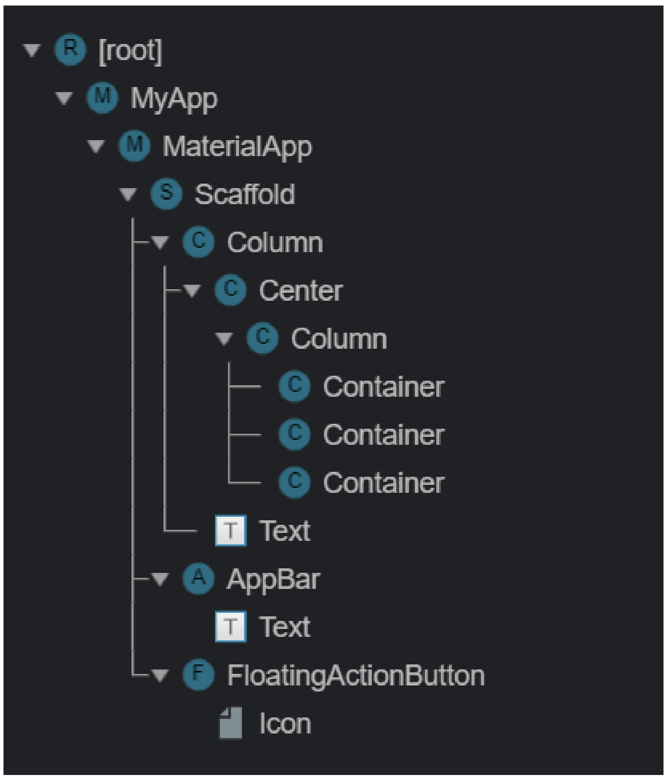

Figure 4: Widget tree

References

- The code written during the session
- Example for stateless widget
- Example for a stateful widget
- Documentation on flutter.dev.
- Some basic widgets
- Containers: cheat sheet
- Rows, Columns and decomposing layouts (the code in this link uses widgets that were not discussed during the session, but is a highly recommended read. Look at this link and this link for details)
- Widget of the week
- Widget of the week, another playlist