Flutter SDK for building

Native iOS and Android apps

05 –Assets, Colors, Text input, and Gestures

Schoology
XCZQ-M7WT-ZG4MC

Dr. Essam Eliwa
essam.eliwa@fa-hists.edu.eg

Fall 2019

Agenda

- **Assets and Displaying images**
- **Colors**
- **Retrieving text and changing state**
- **Gestures**

Assets

- Flutter apps can include both code and *assets* (sometimes called resources).
- An asset is a file that is bundled and deployed with your app, and is accessible at runtime.
- Common types of assets include static data (for example, JSON files), configuration files, icons, and images (JPEG, GIF, PNG, and BMP).

Specifying assets

- Flutter uses the **pubspec.yaml** file, located at the root of your project, to identify assets required by an app.
- To include all assets under a directory, specify the directory name with the / character at the end

```
flutter:
  assets:
    - assets/my_icon.png
    - assets/background.png
```

```
flutter:
  assets:
    - assets/
    - assets/cars/
```

Loading assets

- Your app can access its assets through an AssetBundle object.
- The two main methods on an asset bundle allow you to load a string/text asset (loadString()) or an image/binary asset (load()) out of the bundle

- Each Flutter app has a rootBundle object for easy access to the main asset bundle
- It is possible to load assets directly using the rootBundle global static from package:flutter/services.dart

Loading text assets

```
import 'dart:async' show Future;
import 'package:flutter/services.dart' show rootBundle;

Future<String> loadAsset() async {
    return await rootBundle.loadString('assets/config.json’);
}
```

Loading images

```
Widget build(BuildContext context) {
   return Image(image: AssetImage('graphics/background.png'));
}
```

Declaring resolution-aware image assets

- AssetImage understands how to map a logical requested asset onto one that most closely matches the current device pixel ratio.
- In order for this mapping to work, assets should be arranged according to a particular directory structure.
- The main asset is assumed to correspond to a resolution of 1.0.

```
flutter:
  assets:
    - assets/images/my_icon.png
```

```
.../my_icon.png
.../2.0x/my_icon.png
.../3.0x/my_icon.png
```

Display images

- When using image widgets, we should specify their size using either height or width arguments, or place image widgets in contexts that set tight layout constraints.

- This makes sure that the layout doesn't change when the image is loading. Otherwise, the image may appear to take no space first, then expand to its actual size. It looks like other widgets are pushed away by the image

Display images from network

```
Widget _generateItem(int index) {
    log.info('Generate item $index’);
    return new Container(
                padding: const EdgeInsets.all(5.0),
                child: new Row( children: <Widget>[
                        new Image.network(
                                'http://get.images.com/200x100?text=Item$index’,
                                 width: 200.0, height: 100.0, ),
                         new Expanded(child: new Text('Item $index')) ], ), );
}
```

Colors

- Color and ColorSwatch constants which represent Material design's color palette.
- consider using **Theme.of** to obtain the local ThemeData structure, which exposes the colors selected for the current theme, such as **ThemeData.primaryColor** and **ThemeData.accentColor**
- Most swatches have colors from 100 to 900 in increments of one hundred, plus the color 50.
- **Accent colors** are colors that are used for emphasis in a color scheme. These colors can often be bold or vivid and are used to emphasize or contrast.

Colors

```
Color selection = Colors.green[400];

Color custom1= Color(0xFFCC0FEA);
Color custom2= Color.fromRGBO(66, 165, 245, 0.5);
Color custom3= Color.fromARGP(130,50,60,24);
```

`Color.fromARGP` Construct a color using four integers.

a is the alpha value, with 0 being transparent and 255 being fully opaque.
r is red, from 0 to 255.
g is green, from 0 to 255.
b is blue, from 0 to 255.

Colors.lightGreenAccent[100]	0xFFCCFF90
Colors.lightGreenAccent	**0xFFB2FF59**
Colors.lightGreenAccent[400]	0xFF76FF03
Colors.lightGreenAccent[700]	0xFF64DD17

Colors.lightGreen[50]	0xFFF1F8E9
Colors.lightGreen[100]	0xFFDCEDC8
Colors.lightGreen[200]	0xFFC5E1A5
Colors.lightGreen[300]	0xFFAED581
Colors.lightGreen[400]	0xFF9CCC65
Colors.lightGreen	**0xFF8BC34A**
Colors.lightGreen[600]	0xFF7CB342
Colors.lightGreen[700]	0xFF689F38
Colors.lightGreen[800]	0xFF558B2F
Colors.lightGreen[900]	0xFF33691E

Quiz

- Which of the following colors is **NOT valid**
  A. Colors.blue.shade50
  B. Colors.transparent
  C. Colors(0xFF013487)
  D. Color(0xFFFFFF)
  E. Color(0xFF4CD132)
  F. Color.fromRGBO(66, 165, 245, 1.0);

Answer

- Which of the following colors is **NOT valid**
  A. Colors.blue.shade50
  B. Colors.transparent
  C. Colors(0xFF013487)
  D. Color(0xFFFFFF)
  E. Color(0xFF4CD132)
  F. Color.fromRGBO(66, 165, 245, 1.0);

Text Input

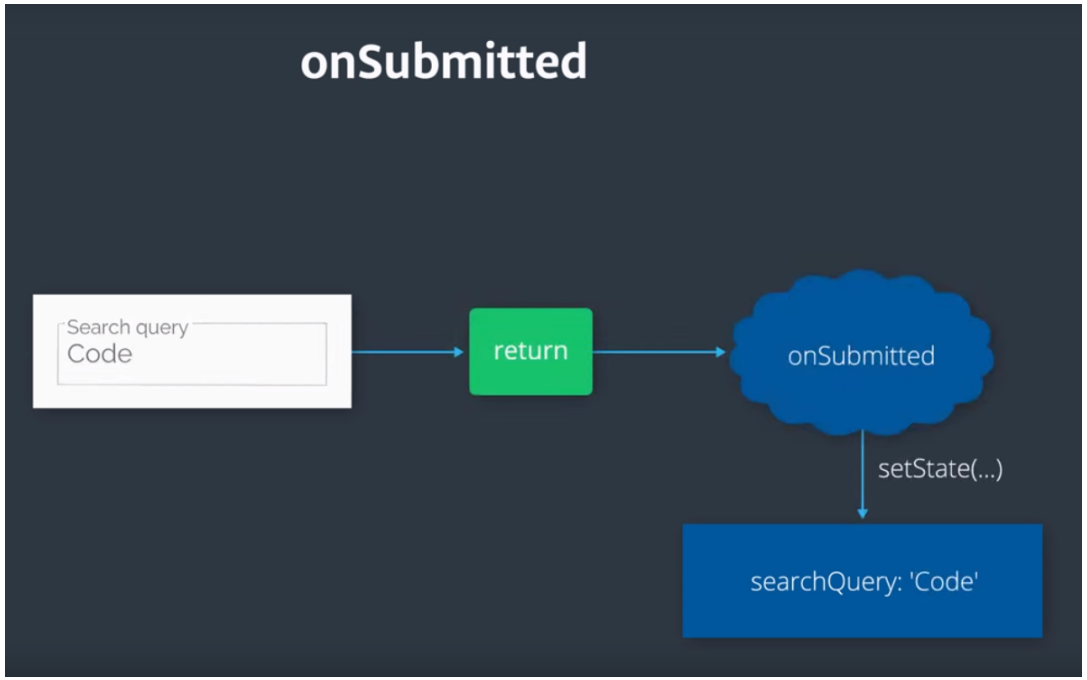
- Text fields can be used to build forms, messaging apps etc.

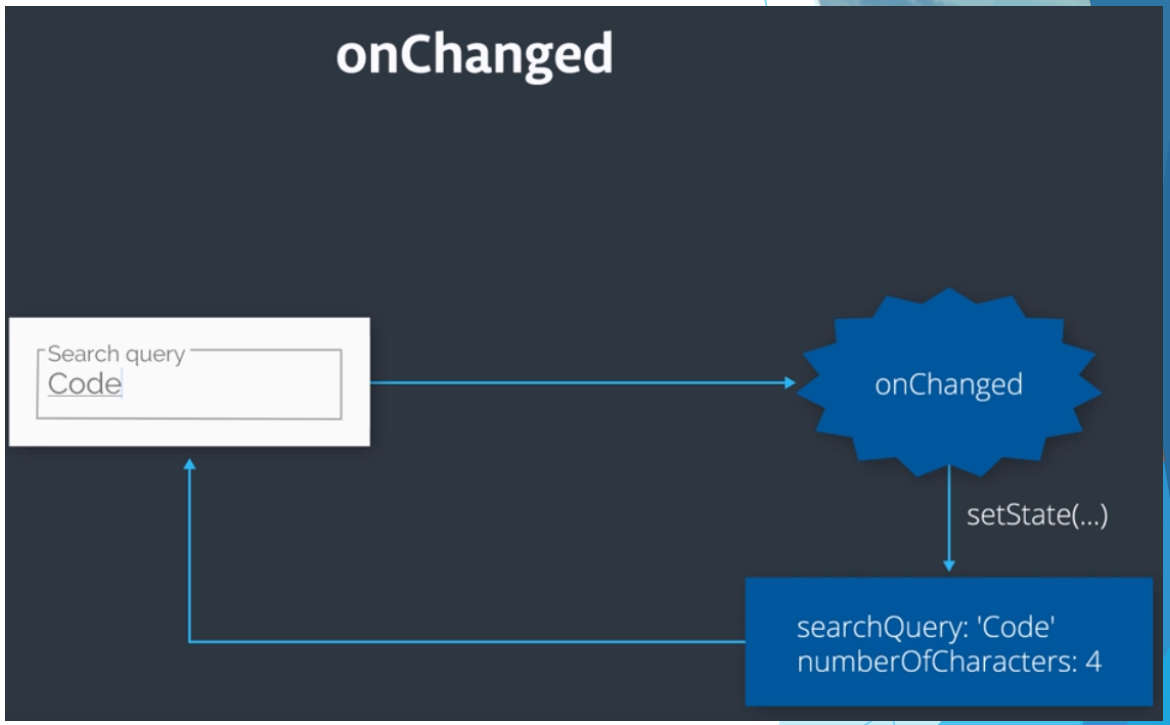
- **TextField** is the most commonly used text input widget.

```
TextField(
    decoration: InputDecoration(
    border: InputBorder.none,
    contentPadding: const EdgeInsets.all(20.0),
    hintText: 'Please enter a search term'
  ),
);
```

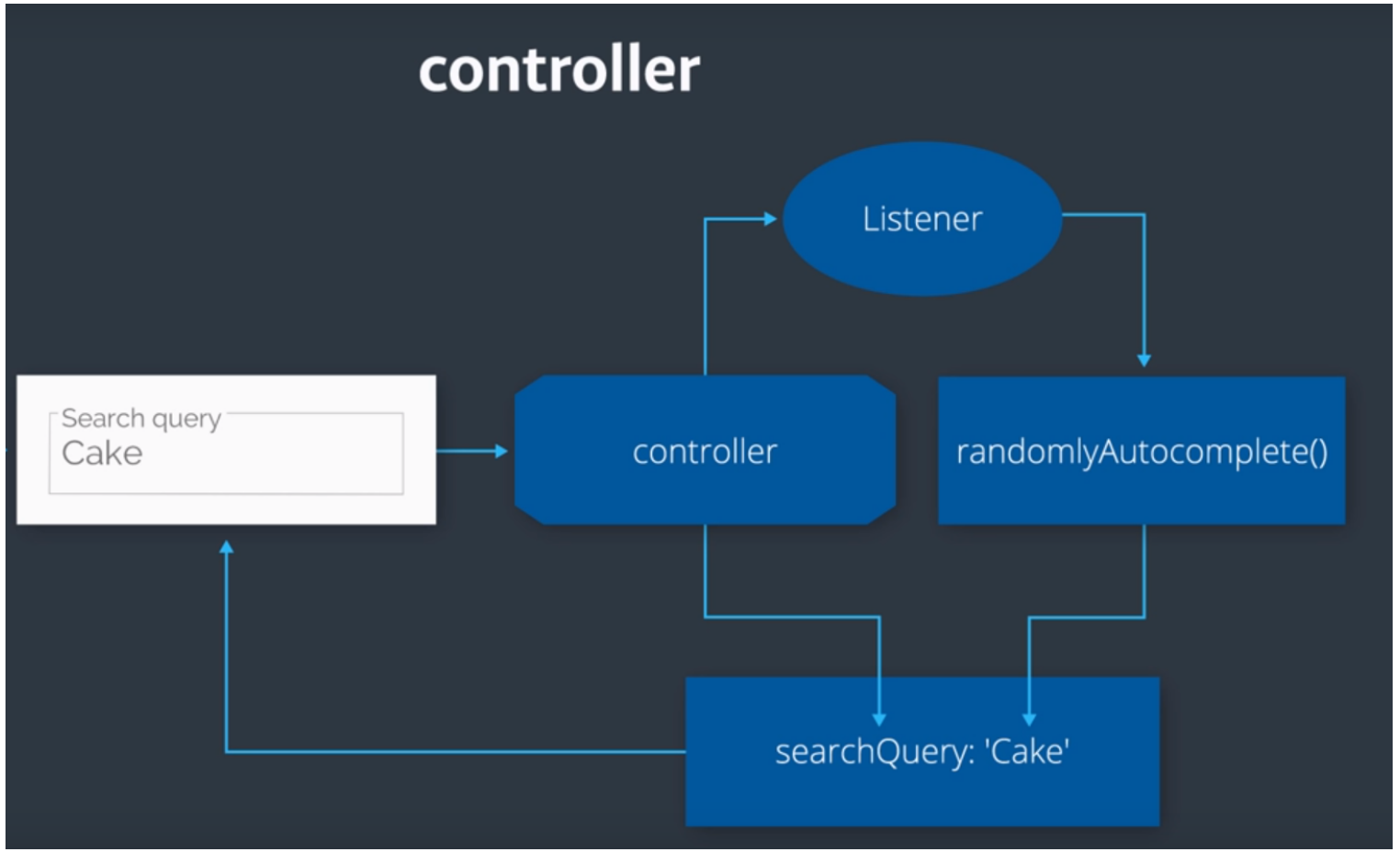
Retrieving text and changing state

- onChanged
- onSubmited
- Controller





TextField Controller



Supply an onChanged callback to a TextField

- This example prints the current value of the text field to the console every time the text changes.

```
TextField(
  onChanged: (text) {
    print("First text field: $text");
  },
  decoration: InputDecoration(
  border: InputBorder.none,
  contentPadding: const EdgeInsets.all(20.0),
  hintText: 'Please enter a search term'
  ),
);
```

Use a TextEditingController

- A more powerful, but more elaborate approach, is to supply a TextEditingController as the controller property of the TextField or a TextFormField.

- To be notified when the text changes, we can listen to the controller using its addListener method.

- Steps
  - Create a TextEditingController
  - Supply the TextEditingController to a TextField
  - Create a function to perform the required action
  - Listen to the controller for changes

Create a TextEditingController

```
class MyCustomForm extends StatefulWidget {
  @override
  _MyCustomFormState createState() => _MyCustomFormState();
}

class _MyCustomFormState extends State<MyCustomForm> {
  // Create a text controller.
 final myController = TextEditingController();

  @override
  void dispose() {
    // Clean up the controller when the Widget is removed from the Widget tree
    myController.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    // We will fill this out in the next step!
  }
}
```

Supply the TextEditingController to a TextField

- In order to work, the TextEditingController must be supplied to either a TextField or a TextFormField. Once it's wired up, we can begin listening for changes to the text field.

```
TextField(
  controller: myController,
);
```

Create a function to print the latest value

- Now, we'll need a function that should run every time the text changes! In this example, we'll create a method that prints out the current value of the text field.

- This method will live inside our _MyCustomFormState class.

```
_printLatestValue() {
  print("text field: ${myController.text}");
}
```

Listen to the controller for changes

- Finally, we need to listen to the TextEditingController and run the _printLatestValue method whenever the text changes. We will use the addListener method to achieve this task.

```
class _MyCustomFormState extends State<MyCustomForm> {
  @override
  void initState() {
    super.initState();

    // Start listening to changes
    myController.addListener(_printLatestValue);
  }

  @override
  void dispose() {
    // Stop listening to text changes
    myController.removeListener(_printLatestValue);

    // Clean up the controller when the Widget is removed from the Widget tree
    myController.dispose();
    super.dispose();
  }
}
```

Gestures in Flutter

Flutter gesture system

- The gesture system in Flutter has two separate layers.
- The first layer has raw **pointer** events, which describe the location and movement of pointers (e.g., touches, mice, and styli) across the screen.
- The second layer has **gestures**, which describe semantic actions that consist of one or more pointer movements.

Pointers

Pointers represent raw data about the user's interaction with the device's screen. There are four types of pointer events:

- **PointerDownEvent** The pointer has contacted the screen at a particular location.
- **PointerMoveEvent** The pointer has moved from one location on the screen to another.
- **PointerUpEvent** The pointer has stopped contacting the screen.
- **PointerCancelEvent** Input from this pointer is no longer directed towards this app.

The pointer down event (and subsequent events for that pointer) are dispatched to the innermost widget found by the hit test. From there, the events bubble up the tree and are dispatched to all the widgets on the path from the innermost widget to the root of the tree.

Gestures

Gestures represent semantic actions (e.g., tap, drag, and scale) that are recognized from multiple individual pointer events. For Example:

- **Tap**
  - **onTapDown** A pointer that might cause a tap has contacted the screen at a particular location.
  - **onTapUp** A pointer that will trigger a tap has stopped contacting the screen at a particular location.
  - **onTap** A tap has occurred.
  - **onTapCancel** The pointer that previously triggered the onTapDown will not end up causing a tap.

Listen to gestures

- To listen to gestures from the widgets layer, use a GestureDetector.

- Many of the Material Components widgets already respond to taps or gestures.

- For example, IconButton and FlatButton respond to presses (taps), and ListView responds to swipes to trigger scrolling.

GestureDetector

- A widget that detects gestures.
- Material design applications typically react to touches with ink splash effects. The InkWell class implements this effect and can be used in place of a GestureDetector for handling taps.

```
GestureDetector(
  onTap: () {
    setState(() { _lights = true; });
  },
  child: Container(
    color: Colors.yellow,
    child: Text('TURN LIGHTS ON'),
  ),
)
```

Refrences

- https://docs.flutter.io/flutter/dart-ui/Color/Color.fromRGBO.html

Questions