Flutter Basics

A basic documentation of Flutter Widgets

Aparna Sriram

3rd Year, CSE

Sri Sairam Engineering College

Last modified: 13th January (Version 1)

Table of Contents

S.No	Topic	Pg.No
1.	Introduction to Flutter	1
2.	Setting up Development Environment	3
3.	Dart Programming Basics	7
4.	Understanding Widgets in Flutter	9
5.	AppBar Widget	13
6.	Row and column Widget	15
7.	Scaffold Widget	17
8.	Image Widget	18
9.	Icon Widget	19
10.	ElevatedButton Widget	20

Introduction to Flutter

Overview on Flutter

Flutter is Google's Mobile SDK for building native iOS, Android, Desktop (Windows, Linux, macOS), and Web apps from a single codebase. It revolves around Widgets, the building blocks of Flutter apps. Widgets are structural elements that come with material design-specific functionalities, and developers can compose new widgets from existing ones through composition. The app's User Interface is seen as a tree of widgets.

Shudder Framework

Shudder is an open-source framework designed to create high-quality, high-performance mobile applications for Android and iOS. It provides a straightforward, robust, efficient, and clear SDK to write mobile apps in Dart, Google's language.

Dart Programming Language

Dart is a programming language developed by Google, primarily designed for building web, mobile, and desktop applications. It is the language used for Flutter development. Dart is known for its simplicity, readability, and efficiency. It supports both Object-Oriented Programming (OOP) and Just-In-Time (JIT) compilation.

Types of Widgets in Flutter

There are two broad categories of widgets in Flutter which are as follows:

- Stateful Widgets
- Stateless Widgets

Why Use Flutter?

- Rich Software Packages: Dart, the programming language behind Flutter, boasts an extensive collection of software packages. This allows developers to easily integrate existing solutions, enhancing the functionalities of their applications.

- Cross-Platform Development: Flutter enables developers to write a single codebase for both Android and iOS applications. This approach promotes code reusability and efficiency in the development process.

- Efficient Testing: With a single codebase, Flutter reduces testing efforts significantly. Automated tests only need to be written once, providing comprehensive coverage for both Android and iOS platforms. This streamlines the testing process and ensures consistent functionality across devices.

- Simplified Development Workflow: Flutter's simplicity accelerates the development process, making it an excellent choice for projects with tight timelines. The straightforward nature of Flutter allows developers to focus on building features rather than dealing with platform-specific complexities.

- Complete Control Over Widgets: Flutter provides developers with complete control over widgets and their layouts. This flexibility allows for a highly customizable user interface, ensuring a seamless and visually appealing user experience.

Differences between Flutter and React Native.

Feature	Flutter	React Native
Language	Dart	JavaScript (React)
Architecture	Reactive framework	Component-based
Performance	High performance (60 fps or 120 fps animation)	Good performance requires a JavaScript bridge
UI Components	Customizable widgets	Native components use bridge for communication
Development Speed	Faster due to hot reload	Fast development with hot reload

Setting up the Development Environment

Android Studio is a popular IDE(integrated development environment) developed by Google to create cross-platform Android applications. First, you must install Android Studio of version 3.0 or later, as it offers an integrated IDE experience for Flutter. You can refer to this for details: Android Studio

Installation of Flutter and Dart Plugins

After the successful installation of Android Studio, you have to install Flutter and Dart plugins. To do so follow the steps mentioned below:

1. Start Android Studio.
2. Open plugin preferences (Configure > Plugins as of v3.6.3.0 or later).
3. Select the Flutter plugin and click Install.
4. Click Yes when prompted to install the Dart plugin.
5. Click Restart when prompted.

Running a Flutter Application

Before running the Flutter application, make sure you have Flutter and Dart installed, and the Flutter plugin is properly configured in your IDE (Android Studio in this case).

Step 1: Open Android Studio

Open Android Studio on your computer.

Step 2: Create a New Flutter Project

1. Click on "Start a new Flutter project" or go to File > New > New Flutter Project.

2. Select "Flutter Application" as the project type and click Next.

3. Verify the Flutter SDK path. If it's not specified, select "Install SDK..." to install it.

4. Enter a project name (e.g., `myapp`) and click Next.

5. Click Finish to create the Flutter project.

Step 3: Configure AVD (Android Virtual Device)

1. In the main toolbar, locate the target selector.

2. Select an Android device for running the app. If none are listed, go to Tools > Android > AVD Manager to create a virtual device.

Step 4: Run the Flutter Application

1. Click the run icon in the toolbar, or go to Run > Run.

2. Wait for the app build to complete.

Step 5: View the App on the Emulator/Device

After the build is complete, the app will run on the selected emulator or connected device. You will see the starter app with the title "Hello World Demo Application" and a welcome message.

Example Code

```
// Importing important packages requires to connect
// Flutter and Dart
import 'package:flutter/material.dart';

// Main Function
void main() {
// Giving command to runApp() to run the app.

/* The purpose of the runApp() function is to attach
the given widget to the screen. */
runApp(const MyApp());
}

// Widget is used to create UI in flutter framework.

/* StatelessWidget is a widget, which does not maintain
any state of the widget. */

/* MyApp extends StatelessWidget and overrides its
build method. */
class MyApp extends StatelessWidget {
const MyApp({Key? key}) : super(key: key);

// This widget is the root of your application.
@override
Widget build(BuildContext context) {
        return MaterialApp(
```

```
        // title of the application
        title: 'Hello World Demo Application',
        // theme of the widget
        theme: ThemeData(
                primarySwatch: Colors.lightGreen,
        ),
        // Inner UI of the application
        home: const MyHomePage(title: 'Home page'),
        );
}
}

/* This class is similar to MyApp instead it
returns Scaffold Widget */
class MyHomePage extends StatelessWidget {
const MyHomePage({Key? key, required this.title}) : super(key: key);
final String title;

@override
Widget build(BuildContext context) {
        return Scaffold(
        appBar: AppBar(
                title: Text(title),
        ),
        // Sets the content to the
        // center of the application page
        body: const Center(
                // Sets the content of the Application
                child: Text(
                'Welcome to GeeksForGeeks!',
        )),
        );
}
}
```

Here's a breakdown of the provided Flutter code:

- The `main` function initializes the Flutter app using `runApp`.
- The `MyApp` class is a `StatelessWidget` representing the root of the application. It sets up the app's theme and the home page.
- The `MyHomePage` class, also a `StatelessWidget`, represents the content of the home page with an app bar and a centred welcome message

Output:

Additional Notes

- Make sure your device or emulator is running and is visible in the AVD Manager.
- If you encounter any issues, check the console for error messages.

Now you've successfully created and run a simple Flutter application! You can continue building and enhancing your app from here.

Dart Programming Basics

Key Features of Dart

- Object-Oriented Paradigm: Dart follows a fully object-oriented paradigm, treating everything as an object. Developers can model real-world entities using classes and objects.

- Strong Typing: Dart is statically-typed, requiring variable types to be declared at compile time. This feature helps catch errors early in the development process.

- Garbage Collection: Automatic garbage collection in Dart simplifies memory management, as the system deallocates memory that is no longer in use.

- Isolates for Concurrency: Dart introduces isolates for concurrent programming. Isolates are independent workers that don't share memory, offering scalability for handling multiple tasks simultaneously.

- Asynchronous Programming: Dart supports asynchronous programming through Future and Stream classes, essential for non-blocking operations like I/O or network requests.

Dart Syntax Basics

Hello World Program:

```
void main() {
  print('Hello, Dart!');
}
```

Dart programs start execution from the main function.

Variables:

```
String name = 'John';
int age = 28;
double salary = 50000.75;
bool isStudent = false;
```

Dart uses explicit type annotations for variable declarations.

Functions:

```
int add(int a, int b) {
  return a + b;
}
```

Dart functions are defined using the functionName(parameters) { body } syntax.

Control Flow:

```
if (isStudent) {
  print('Student is valid.');
} else {
  print('Invalid student.');
}

for (int i = 0; i < 5; i++) {
  print('Iteration $i');
}

while (age > 0) {
  print('Age is $age');
  age--;
}
```

Dart supports common control flow structures like if-else statements, for and while loops.

Collections:

```
List<int> numbers = [1, 2, 3, 4, 5];
Map<String, dynamic> person = {'name': 'Aparna', 'age': 20, 'isEmployee': false,'isIntern': true};
```

Dart provides built-in support for lists and maps.

Understanding Widgets in Flutter

In Flutter, widgets are the fundamental building blocks for creating user interfaces. They are elements that describe the structure and appearance of the UI in a Flutter app. Each visual element on the screen, such as buttons, text, images, etc., is represented by a widget. The entire structure of a Flutter app is essentially a tree of widgets.

Categories of Widgets:

1. Accessibility: Widgets that enhance the accessibility of a Flutter app.
2. Animation and Motion: Widgets responsible for adding animation to other widgets.
3. Assets, Images, and Icons: Widgets that handle assets, display images, and show icons.
4. Async: Widgets providing asynchronous functionality in the app.
5. Basics: Essential widgets necessary for Flutter app development.
6. Cupertino: iOS-styled widgets.
7. Input: Widgets providing input functionality.
8. Interaction Models: Widgets managing touch events and routing users to different views.
9. Layout: Widgets for placing other widgets on the screen as needed.
10. Material Components: Widgets following Google's material design.
11. Painting and Effects: Widgets applying visual changes without changing layout or shape.
12. Scrolling: Widgets enabling scrollability to non-scrollable widgets.
13. Styling: Widgets dealing with app theme, responsiveness, and sizing.
14. Text: Widgets for displaying text.

Types of Widgets:

Stateless Widget: A widget that doesn't maintain any state. Its appearance is purely based on the configuration and state passed to it.

Stateful Widget: A widget that can maintain state. It can change its appearance based on state changes.

Stateless Widget Example

```
import 'package:flutter/material.dart';

void main() => runApp(const GeeksforGeeks());

class GeeksforGeeks extends StatelessWidget {
  const GeeksforGeeks({Key? key}) : super(key: key);

  @override
```

```
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        backgroundColor: Colors.lightGreen,
        appBar: AppBar(
          backgroundColor: Colors.green,
          title: const Text("GeeksforGeeks"),
        ),
        body: Container(
          child: const Center(
            child: Text("Hello Geeks!!"),
          ),
        ),
      ),
    );
  }
}
```

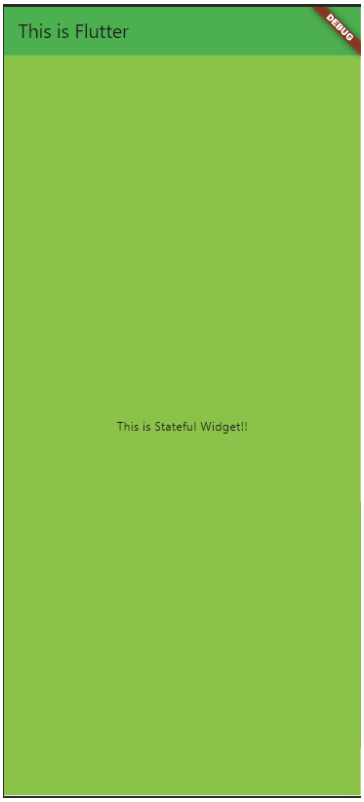
Output:

Stateful Widget Example:

```
import 'package:flutter/material.dart';

void main() => runApp(const MyApp());
```

```
class MyApp extends StatefulWidget {
  const MyApp({Key? key}) : super(key: key);

  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        backgroundColor: Colors.lightGreen,
        appBar: AppBar(
          backgroundColor: Colors.green,
          title: const Text("GeeksforGeeks"),
        ),
        body: const Center(
          child: Text("Hello Geeks!!"),
        ),
      ),
    );
  }
}
```

Output:



In these examples, the Scaffold, AppBar, Text, Container, and Center are all different types of widgets used to compose the UI. The StatelessWidget and StatefulWidget represent the two types of widgets based on whether they maintain state or not.

AppBar Widget

The Material Design AppBar is a fundamental element in Flutter applications, consisting of a toolbar and additional widgets like TabBar and FlexibleSpaceBar. It is commonly used within the Scaffold.appBar property, providing a fixed-height widget at the top of the screen.

- The AppBar comprises a toolbar with common actions, including IconButtons and a PopupMenuButton for less frequent operations.
- Actions are typically displayed on the top right, with the title centred between them.
- The bottom section is often reserved for a TabBar.
- A flexible space widget can be specified to appear behind the toolbar and bottom widgets.

Use Case:
The AppBar is used to create a top-level navigation and actions space in the app, providing quick access to common functionalities and allowing for easy navigation.

Example code:

```
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: DefaultTabController(
        length: 2, // Specify the number of tabs
        child: Scaffold(
          appBar: AppBar(
            leading: IconButton(
              icon: Icon(Icons.menu),
              onPressed: () {
                // Handle drawer or navigation menu opening
              },
            ),
            title: Text('My App'),
            actions: [
              IconButton(
                icon: Icon(Icons.search),
                onPressed: () {
                  // Handle search action
```

```
          },
        ),
        IconButton(
          icon: Icon(Icons.settings),
          onPressed: () {
            // Navigate to settings page
          },
        ),
      ],
      bottom: TabBar(
        tabs: [
          Tab(text: 'Tab 1'),
          Tab(text: 'Tab 2'),
        ],
      ),
    ),
    body: TabBarView(
      children: [
        // Contents of Tab 1
        Center(
          child: Text('Tab 1 Content'),
        ),
        // Contents of Tab 2
        Center(
          child: Text('Tab 2 Content'),
        ),
      ],
    ),
  ),
 );}
```

Explanation:

AppBar: A top bar with a menu icon, app title ('My App'), and action icons for search and settings. It also includes a bottom TabBar with two tabs.

Tabs: Two tabs ('Tab 1' and 'Tab 2') for navigation.

Tab Content: Content for each tab is displayed in a TabBarView. In this example, it's just text saying 'Tab 1 Content' and 'Tab 2 Content'.

Navigation: The menu icon and tabs provide a basic navigation structure for the app.

Row and Column Widget

In Flutter, Row and Column are layout widgets that allow you to arrange child widgets horizontally (for Row) or vertically (for Column). Here's a brief explanation of each:

Row Widget:

- **Description:** The Row widget in Flutter creates a horizontal array of children widgets.
- **Alignment Properties:** You can align the children within the row using properties like mainAxisAlignment and crossAxisAlignment.
- **mainAxisAlignment**: Defines how the children should be placed along the main axis (horizontal axis in the case of Row).
- **crossAxisAlignment:** Determines how the children should be aligned along the cross axis (vertical axis in the case of Row).

Common Use Cases:

- Creating a horizontal list of items.
- Arranging items side by side.

Column Widget:

- **Description:** The Column widget in Flutter creates a vertical array of children widgets.
- **Alignment Properties:** Similar to Row, you can use properties like mainAxisAlignment and crossAxisAlignment to align the children within the column.
- **mainAxisAlignment**: Specifies how the children should be placed along the main axis (vertical axis in the case of Column).
- **crossAxisAlignment:** Defines how the children should be aligned along the cross axis (horizontal axis in the case of Column).

Common Use Cases:

- Creating a vertical list of items.
- Arranging items one below the other.

Example Code:

```
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Row and Column Example'),
```

```
      ),
      body: Center(
        child: Row(
          mainAxisAlignment: MainAxisAlignment.spaceEvenly,
          children: [
            Icon(Icons.star, size: 50, color: Colors.yellow),
            Icon(Icons.star, size: 50, color: Colors.yellow),
            Icon(Icons.star, size: 50, color: Colors.yellow),
          ],
        ),
      ),
    ),
  );
 }
}
```

In this example:

- The app has a single screen with an AppBar titled 'Row and Column Example'.
- The body of the screen contains a Row widget with three yellow star icons.
- The mainAxisAlignment property of Row is set to spaceEvenly, which distributes the icons evenly along the horizontal axis.

Scaffold Widget

Scaffold is a fundamental and versatile widget in Flutter used to create the basic structure of a visual interface for mobile applications. It provides a layout scaffold that includes common elements such as an AppBar, BottomNavigationBar, and a body for the main content.

Here's a concise breakdown:

- **AppBar**: The top app bar typically contains a title, and it can also hold icons, buttons, or other widgets. It acts as a header for the screen.
- **Body:** The main content of the screen is placed within the body property. It can be any widget or a combination of widgets arranged using layout widgets like Column or ListView.
- **FloatingActionButton:** An optional floating action button can be added to perform a primary action on the screen. It is placed at the bottom right by default.
- **BottomNavigationBar:** If navigation between different sections or pages is required, a bottom navigation bar can be included, allowing users to switch between views.
- **Drawer:** A sliding drawer (side menu) can be added on the left or right side of the screen to provide additional navigation options.

Example Code:

```
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('My App'),
        ),
        body: Center(
          child: Text('Hello, Flutter!'),
        ),
      ), );  }}
```

Description:

- The app has a Scaffold with a simple AppBar containing the title 'My App'.
- The main content of the screen is placed in the body, which consists of a centred text widget saying 'Hello, Flutter!'.

Image Widget

The Image widget in Flutter is used to display images within your app. It supports various sources, including network images, assets, and memory. Here's a concise breakdown:

Example Code:

```
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Image Widget Example'),
        ),
        body: Center(
          child: Image.network(
            'https://example.com/image.jpg',
            width: 150,
            height: 150,
            fit: BoxFit.cover,
          ),
        ),
      ),
    );} }
```

Description:

Import Statement: Import the necessary material library.
Image Widget: Use the Image widget with the network constructor to load an image from a URL.
Width and Height: Set the width and height properties to define the dimensions of the image.
Fit Property: Use fit: BoxFit.cover to ensure the image covers the allotted space, adjusting its size as needed.
Scaffold Structure: The Image widget is placed within the Scaffold body, creating a basic structure for displaying an image in the app.

Icon Widget

The `Icon` widget in Flutter is used to display various icons within your app. It supports a wide range of predefined icons from the Material Icons library and allows customization. Here's a concise breakdown:

Example Code:

```
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}
class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('Icon Widget Example'),
        ),
        body: Center(
          child: Icon(
            Icons.star,
            size: 50,
            color: Colors.yellow,
          ),
        ),
      ),
    );}}
```

Description:

1. **Import Statement:** Import the necessary material library.
2. **Icon Widget:** Use the `Icon` widget with the `Icons.star` constructor to display a star icon.
3. **Size Property:** Set the `size` property to define the size of the icon.
4. **Color Property:** Use the `colour` property to specify the colour of the icon.
5. **Scaffold Structure:** The `Icon` widget is placed within the `Scaffold` body, creating a basic structure for displaying an icon in the app.

ElevatedButton Widget

The `ElevatedButton` widget in Flutter is used to create a raised button, which typically represents a primary or important action in your application. Here's a concise breakdown along with a short example:

Example Code:

```
import 'package:flutter/material.dart';

void main() {
  runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: Text('ElevatedButton Example'),
        ),
        body: Center(
          child: ElevatedButton(
            onPressed: () {
              // Add your button press logic here
            },
            child: Text('Press Me'),
          ),
        ),
      ),
    );
  }
}
```

Description:

1. **Import Statement:** Import the necessary material library.

2. **ElevatedButton Widget**: Use the `ElevatedButton` widget to create a raised button.

3. **onPressed Property:** Assign a callback function to the `onPressed` property to define the action to be executed when the button is pressed.

4. **child Property:** Set the `child` property to specify the text or widget displayed on the button. In this example, it's a `Text` widget with the label 'Press Me'.

5. **Scaffold Structure:** The `ElevatedButton` is placed within the `Scaffold` body, creating a basic structure for displaying a raised button in the app.

The Flutter widgets covered in this document represent just a glimpse into the vast capabilities of the Flutter framework. Flutter stands out for offering an extensive collection of building blocks, known as widgets, providing developers with the tools to craft intricate and feature-rich user interfaces. Leveraging Flutter's flexibility and robust widget library, developers can architect applications that cater to diverse design requirements, delivering a wide array of user experiences. As you delve deeper into Flutter development, you'll discover the richness and versatility of the framework, enabling you to bring your app ideas to life with creativity and efficiency.