2024

Cross Platform Application Development

Dr. Babasaheb Ambedkar Open University

Course Writer

Dr. Himanshu Patel
Assistant Professor, School of Computer Science,
Dr. Babasaheb Ambedkar Open University, Ahmedabad

Subject Reviewer

Mrs. Vishakha Patel
Team Leader, SWISS Infotech, Bhavnagar

Editors

Prof. (Dr.) Nilesh Modi
Professor and Director, School of Computer Science,
Dr. Babasaheb Ambedkar Open University, Ahmedabad

Dr. Himanshu Patel
Assistant Professor, School of Computer Science,
Dr. Babasaheb Ambedkar Open University, Ahmedabad

Acknowledgement: “The content in this book is modifications based on work created and shared by the Flutter documentation (https://docs.flutter.dev/) and used according to terms described in the Creative Commons 4.0 Attribution License.”

ISBN-

Printed and published by: Dr. Babasaheb Ambedkar Open University, Ahmedabad

Dr. Babasaheb Ambedkar Open University

MCA-303

Cross Platform Application Development

Block-1

Introduction to Flutter and Dart

Unit-1: Introduction to Cross Platform Application Development

1

Unit Structure

1.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Define: Cross Platform Applicaation Development
- Know History of Flutter application development
- What is Flutter? and List Key Features of Flutter
- Enumerate advantate, disadvantage and applications of Flutter
- Describe Flutter features and Understand Flutter architecture
- Enumerate factors that affect Mobile Application development

1.2 CROSS PLATFORM APPLICATION DEVELOPMENT

Cross-platform development is the practice of creating software applications that can run on multiple operating systems or platforms (such as Android, iOS, Windows, macOS, Linux, etc.) using a single codebase. Instead of writing separate versions of an app for each platform, cross-platform development allows developers to write the code once and deploy it across multiple environments with minimal changes.

Key Benefits of Cross-Platform Development:

1. **Code Reusability**: Developers can reuse a large portion of the code across different platforms, saving time and effort.
2. **Cost Efficiency**: It reduces the cost of development since one team can handle both iOS and Android (or other platforms) rather than separate teams for each.
3. **Consistent User Experience**: With a shared codebase, it’s easier to ensure that the app provides a consistent experience across platforms.
4. **Faster Time to Market**: Since the code is written once and deployed across platforms, the development cycle is shorter.
5. **Easier Maintenance**: Any updates or bug fixes can be made in one place and applied across all platforms simultaneously.

Popular Cross-Platform Development Frameworks:

1. **Flutter**: Developed by Google, it uses Dart and provides native performance, expressive UIs, and a fast development cycle.

2. **React Native**: Built by Facebook, it uses JavaScript and allows developers to build mobile apps that work across platforms.
3. **Xamarin**: Owned by Microsoft, it uses C# and enables code sharing across iOS, Android, and Windows.
4. **Ionic**: Based on Angular, it allows building hybrid mobile apps with web technologies.

How It Works:

There are two working models used by Cross-platform frameworks:

1. **Abstraction Layers**: Cross-platform frameworks provide an abstraction layer between the app's code and the underlying platform APIs, allowing the app to work on different platforms without having to reimplement platform-specific code.
2. **Native and Hybrid Approaches**: Some frameworks (like React Native and Flutter) provide near-native performance by compiling to native code, while others (like Ionic) are more web-based and rely on a browser environment within the app.

Challenges:

Major challenges faced by the Cross-platform application development are:

- **Performance**: Some cross-platform frameworks may not achieve the same level of performance as fully native apps, especially in complex or resource-intensive applications.
- **Platform-Specific Features**: While most frameworks provide access to common device features, developers may need to write platform-specific code or plugins for some unique functionalities.
- **UI Consistency**: Ensuring a native-like look and feel across different platforms can sometimes require additional work.

1.3 INTRODUCTION TO FLUTTER

Flutter is an open-source UI software development kit (SDK) created by Google. It is used to develop cross-platform applications from a single codebase. This means that with Flutter, you can build apps for Android, iOS, web, desktop (Windows, macOS, Linux), and even embedded devices, all using the same codebase.

Flutter is a framework that allows developers to build natively compiled applications across multiple platforms using a single programming language, Dart. The framework includes a rich set of pre-designed widgets, tools, and libraries that help developers create visually appealing and highly responsive applications. The use of widgets in Flutter is central to its design, making it easy to customize and create a consistent look and feel across different platforms.

Flutter represents a significant shift in how developers can create applications across multiple platforms. By providing a single codebase that works on mobile, web, desktop, and embedded systems, Flutter offers a versatile, efficient, and powerful solution for modern app development. Whether you are a beginner or an experienced developer, learning Flutter can open up new opportunities and streamline your development process.

Overall, Flutter is popular among developers for its ability to simplify the process of building cross-platform apps, reducing development time and effort while delivering high-quality, natively compiled applications.

1.4 HISTORY OF FLUTTER

Flutter has an interesting history that reflects Google's efforts to streamline and modernize app development across multiple platforms. Here's an overview:

1. Initial Development and Announcement (2015-2017)

- **2015:** Flutter began as an internal project at Google under the codename "Sky." The initial goal was to create a platform that could render consistently at 120 frames per second on Android.
- **May 2017:** Flutter was officially announced at Google I/O 2017. At this stage, it was still in an early alpha version. Google introduced Flutter as a way to build high-performance, high-fidelity mobile apps for iOS and Android from a single codebase.

2. Beta and Preview Releases (2018)

- **February 2018:** Flutter reached its first beta release. This was a significant milestone as it marked the SDK's readiness for broader developer adoption. During this time, Google emphasized the benefits of Flutter's fast development cycle, expressive and flexible UI, and native performance.
- **May 2018:** At Google I/O 2018, Google announced the release of Flutter Beta 3. This version included support for Visual Studio Code, new widgets, and improved documentation, making it more accessible and feature-rich.

3. Stable Release (2018)

- **December 4, 2018:** Flutter 1.0 was released at the Flutter Live event in London. This marked the official stable release of Flutter, making it production-ready for developers to use in building commercial apps. The launch was accompanied by a growing ecosystem of packages and plugins, enabling developers to integrate various functionalities into their apps easily.

4. Expanding to Web and Desktop (2019-2020)

- **May 2019:** At Google I/O 2019, Google introduced "Flutter for Web," a technical preview that extended Flutter’s capabilities to web applications. This was part of a broader strategy to make Flutter a truly cross-platform framework.
- **2020:** Google further expanded Flutter to include support for desktop platforms (Windows, macOS, and Linux). These developments were part of the "Flutter Everywhere" initiative, aiming to provide a unified development experience across all major platforms.

5. Flutter 2.0 and Beyond (2021-Present)

- **March 3, 2021:** Google released Flutter 2.0, a significant update that included stable support for web applications and an improved experience for desktop apps. This version also introduced "Flutter DevTools," providing developers with better tools for debugging and performance optimization.
- **September 2021:** Flutter 2.5 was released, bringing performance improvements, better DevTools, and updated material design components. Google also began focusing on improving the developer experience with better documentation and community support.

6. Recent Developments (2022-Present)

- Flutter continues to evolve with regular updates, introducing new features, performance enhancements, and expanding support for new platforms and devices. The community and ecosystem around Flutter have grown significantly, making it one of the most popular frameworks for cross-platform app development.

Flutter's history reflects its evolution from an experimental project to a fully-fledged framework that is now widely adopted for building applications across a variety of platforms.

1.5 KEY FEATURES OF FLUTTER

1. **Dart Language**: Flutter apps are written in Dart, a programming language also developed by Google. Dart is designed for fast and efficient app development.
2. **Widgets**: Flutter is based on a rich set of customizable widgets, which are the building blocks of the user interface. Everything in Flutter is a widget, from buttons to padding to layouts.
3. **Hot Reload**: Flutter offers a "hot reload" feature, allowing developers to see changes in real-time without restarting the app, making the development process faster and more efficient.
4. **Performance**: Flutter apps are compiled directly to native ARM code, meaning they can achieve high performance similar to native apps.
5. **Customizable UI**: Flutter allows for highly customizable and flexible UI designs, thanks to its widget-based architecture. Developers can create visually rich, custom user interfaces that look consistent across different platforms.
6. **Strong Community and Ecosystem**: Being an open-source project, Flutter has a large and active community, with many third-party packages and plugins available to extend its functionality.
7. **Skia Graphics Engine**: Flutter uses the Skia graphics engine to render visuals, which ensures high-quality and smooth animations across all platforms.

1.6 ADVANTAGES OF FLUTTER

Following are the major advantages of Flutter framework

Cross-Platform Development:

- **Single Codebase:** With Flutter, developers can write a single codebase and deploy it across multiple platforms (iOS, Android, Web, Desktop). This significantly reduces development time and effort.
- **Consistent UI/UX:** Flutter provides a consistent look and feel across platforms, ensuring a uniform user experience.

High Performance:

- **Native Compilation:** Flutter compiles to native ARM code, which ensures high performance and fast execution on mobile devices.
- **GPU Rendering:** The framework uses the Skia graphics engine, enabling smooth and fast UI rendering, even for complex animations.

Hot Reload:

- **Rapid Iteration:** Flutter's hot reload feature allows developers to see changes instantly without restarting the app. This accelerates development, debugging, and experimentation.

Rich Set of Widgets:

- **Customizable UI Components:** Flutter comes with a wide range of pre-designed widgets that are highly customizable, enabling developers to create complex and visually appealing user interfaces.
- **Platform-Specific Design:** Flutter's widgets adapt to the design conventions of the target platform, such as Material Design for Android and Cupertino for iOS.

Strong Community and Ecosystem:

- **Open-Source:** Being an open-source project, Flutter has a large and active community, which contributes to a growing ecosystem of libraries, tools, and resources.

- **Extensive Documentation:** Flutter offers comprehensive documentation, tutorials, and a supportive community, making it easier for developers to learn and solve problems.

Flexibility and Customization:

- **Custom Animations:** Flutter provides powerful tools for creating custom animations and transitions, allowing developers to build engaging user experiences.
- **Third-Party Integrations:** Flutter supports integration with a wide range of third-party services and libraries, expanding its capabilities.

Check Your Progress-1

a) What is cross-platform application development?

b) What is Flutter?

c) What is latest version of Flutter?

d) Flutter is an open-source UI software development kit (SDK) created by Google. (True/False)

e) React Native framwork is built by Microsoft.

f) Xamarin is Based on Angular, it allows building hybrid mobile apps with web technologies. (True/False)

g) Flutter offers a "hot reload" feature, allowing developers to see changes in real-time without restarting the app (True/False)

1.7 DISADVANTAGES OF FLUTTER

Flutter is a powerful and flexible framework that offers many advantages for cross-platform development, particularly for mobile applications. However, it also has some limitations, such as larger app sizes and potential challenges in managing complex projects. When considering Flutter, it's essential to weigh these pros and cons based on the specific needs and goals of your project.

Large App Size:

- **Heavy Initial Package:** Flutter apps tend to have a larger binary size compared to native apps due to the inclusion of the Flutter engine and framework libraries. This can be a concern for users with limited storage space.

Limited Native APIs:

- **Platform-Specific Features:** While Flutter provides access to many native features, some platform-specific APIs might require additional work, such as writing native code or using third-party plugins.
- **Lag in New Features:** New features and APIs released by iOS or Android may take time to be fully supported in Flutter.

Complexity for Larger Projects:

- **Scalability Issues:** For very large and complex applications, managing a single codebase across multiple platforms can become challenging, especially if there are significant differences in platform requirements.
- **State Management:** Managing state in Flutter can become complex in large applications, requiring careful planning and use of state management libraries.

Learning Curve:

- **Dart Language:** Flutter uses Dart, which is not as widely adopted as other languages like JavaScript, Swift, or Kotlin. Developers unfamiliar with Dart may face a learning curve.
- **Custom Development Practices:** Flutter's approach to UI design and development, which heavily relies on widgets, may require developers to adapt to new design patterns and practices.

Limited Web and Desktop Support:

- **Maturity Level:** While Flutter has expanded to support web and desktop applications, these platforms are still not as mature as mobile support, leading to potential limitations or bugs.
- **Performance Issues:** Web and desktop apps built with Flutter might not yet match the performance and feature parity of those built with native or other mature web/desktop frameworks.

1.8 APPLICATIONS OF FLUTTER

Cross-platform development is ideal for businesses or developers who want to reach a wide audience quickly and efficiently while minimizing the effort needed to manage different versions of an app.Flutter is used by a wide range of companies and developers for various types of applications, including:

- **Mobile Applications:** With Flutter, developers can create high-performance mobile applications for both Android and iOS.
- **Web Applications:** Flutter for Web allows developers to create web apps with the same codebase used for mobile apps.
- **Desktop Applications:** Flutter also supports the development of desktop applications for Windows, macOS, and Linux.
- **Embedded Applications:** Flutter can be used for embedded devices, offering the potential for a wide range of IoT applications.

1.9 FLUTTER ARCHITECTURE

Flutter is a cross-platform UI toolkit that is designed to allow code reuse across operating systems such as iOS and Android, while also allowing applications to interface directly with underlying platform services. The goal is to enable developers to deliver high-performance apps that feel natural on different platforms, embracing differences where they exist while sharing as much code as possible.

During development, Flutter apps run in a VM that offers stateful hot reload of changes without needing a full recompile. For release, Flutter apps are compiled directly to machine code, whether Intel x64 or ARM instructions, or to JavaScript if targeting the web. The framework is open source, with a permissive BSD license, and has a thriving ecosystem of third-party packages that supplement the core library functionality.

Architectural layers

Flutter is designed as an extensible, layered system. It exists as a series of independent libraries that each depend on the underlying layer. No layer has privileged access to the layer below, and every part of the framework level is designed to be optional and replaceable.

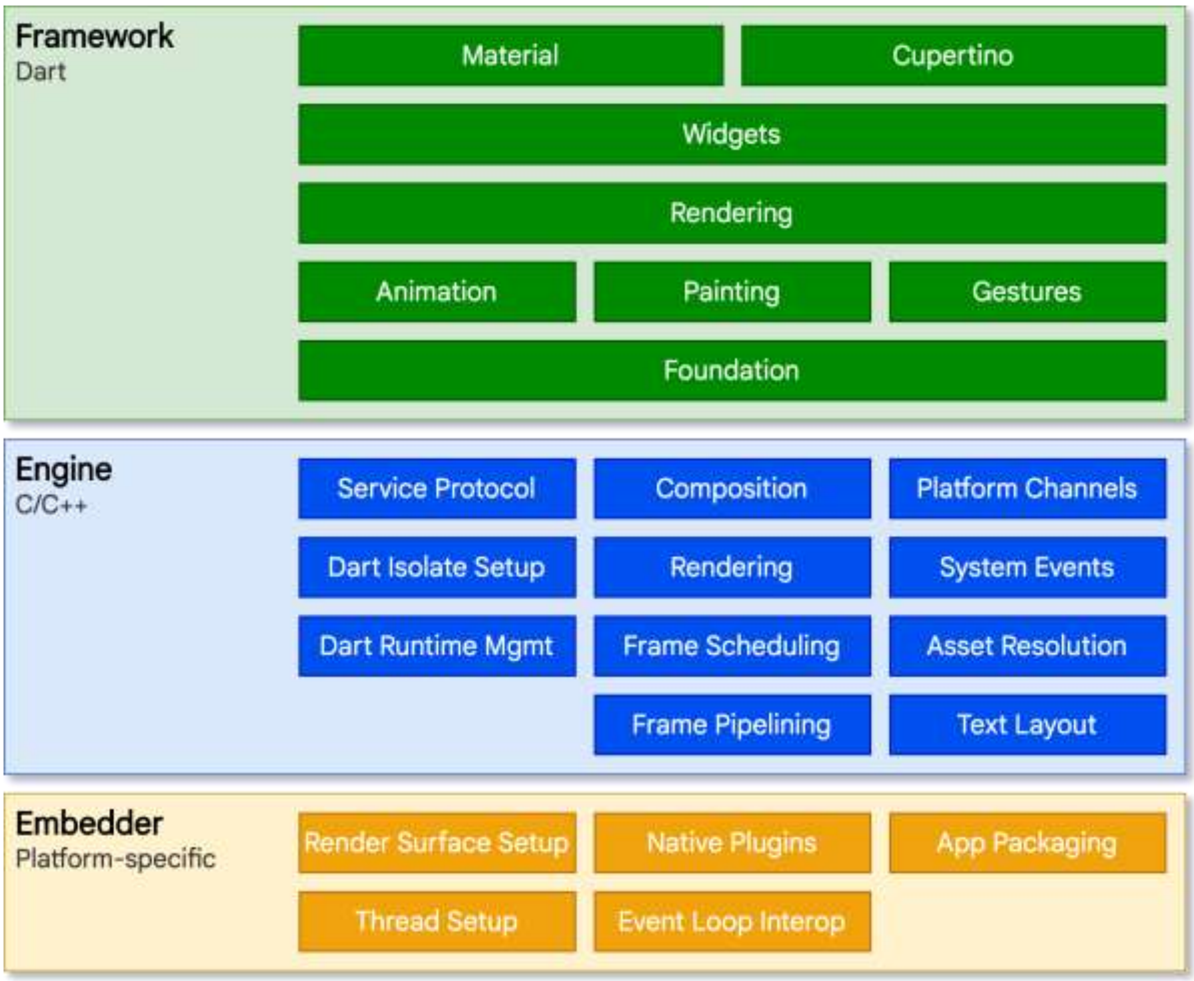


Figure-1: Flutter Architectural Layer

To the underlying operating system, Flutter applications are packaged in the same way as any other native application. A platform-specific embedder provides an entrypoint; coordinates with the underlying operating system for access to services like rendering surfaces, accessibility, and input; and manages the message event loop. The embedder is written in a language that is appropriate for the platform: currently Java and C++ for Android, Objective-C/Objective-C++ for iOS and macOS, and C++ for Windows and Linux. Using the embedder, Flutter code can be integrated into an existing application as a module, or the code might be the entire content of the application. Flutter includes a number of embedders for common target platforms, but other embedders also exist.

At the core of Flutter is the **Flutter engine**, which is mostly written in C++ and supports the primitives necessary to support all Flutter applications. The engine is responsible for rasterizing composited scenes whenever a new frame needs to be painted. It provides the low-level implementation of Flutter's core API, including graphics (through Impeller on iOS and coming to Android and macOS, and Skia on other platforms) text layout, file and network I/O, accessibility support, plugin architecture, and a Dart runtime and compile toolchain.

The engine is exposed to the Flutter framework through dart:ui, which wraps the underlying C++ code in Dart classes. This library exposes the lowest-level primitives, such as classes for driving input, graphics, and text rendering subsystems.

Typically, developers interact with Flutter through the **Flutter framework**, which provides a modern, reactive framework written in the Dart language. It includes a rich set of platform, layout, and foundational libraries, composed of a series of layers. Working from the bottom to the top, we have:

- Basic **foundational** classes, and building block services such as **animation, painting, and gestures** that offer commonly used abstractions over the underlying foundation.
- The **rendering layer** provides an abstraction for dealing with layout. With this layer, you can build a tree of renderable objects. You can manipulate these objects dynamically, with the tree automatically updating the layout to reflect your changes.
- The **widgets layer** is a composition abstraction. Each render object in the rendering layer has a corresponding class in the widgets layer. In addition, the widgets layer allows you to define combinations of classes that you can reuse. This is the layer at which the reactive programming model is introduced.
- The **Material** and **Cupertino** libraries offer comprehensive sets of controls that use the widget layer's composition primitives to implement the Material or iOS design languages.

The Flutter framework is relatively small; many higher-level features that developers might use are implemented as packages, including platform plugins like camera and webview, as well as platform-agnostic features like characters, http, and animations that build upon the core Dart and Flutter libraries. Some of these packages come from the broader ecosystem, covering services like in-app payments, Apple authentication, and animations.

Check your progress-2

a) Flutter is designed as an extensible, layered system (True / False)

b) The Flutter framework is relatively large

c) The widgets layer is a composition abstraction

1.10 LET US SUM UP

In this unit we have discussed what is cross platform application development and list some popular framework for it. You have got detailed understanding of Flutter and its key features, advantages and disadvantages of Flutter, and we also discuss overview of Flutter architecture.

1.11 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a See Page-3, 1-b See Page-4

1-b the latest version of Flutter is Flutter 3.24, released in 2024.

1-c True, 1-d True, 1-e False, 1-f False, 1-g True

2-a True, 2-b False, 2-c True

1.12 FURTHER READING

- https://docs.flutter.dev/

1.13 ASSIGNMENTS

1 What is flutter?

2 Explain key features of flutter.

3 Explain History of Flutter

4 Explain architecture of Flutter in details

5 List advantage, disadvantage and applications of flutte

Unit-2: Introduction to Dart Language 2

Unit Structure

2.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Explain why Dart is used for Flutter development and how it contributes to fast development, high performance, and flexible UI rendering.
- Run basic Dart programs using DartPad or a local setup and understand the structure of a Dart application.
- Work with Dart's basic syntax, including variables, types (int, double, String, bool), and data structures like lists and maps.
- Implement control flow statements such as conditionals (if, else, switch) and loops (for, while) to create logic in Dart programs.
- Define and call functions in Dart, including using optional and named parameters to create more flexible and readable code.
- Create classes, objects, and constructors in Dart, and apply inheritance and polymorphism to build reusable and organized code.
- Use Dart's Future, async, and await to manage asynchronous operations, such as fetching data or executing time-consuming tasks.
- Understand how Dart integrates with Flutter, including how widgets are written in Dart, and how asynchronous programming is used in real-world Flutter apps.

2.2 INTRODUCTION

Dart is the programming language behind Flutter, designed by Google for modern web, mobile, and desktop development. Though Flutter is Dart's most popular use case, Dart is a general-purpose, object-oriented language that powers Flutter's reactive programming model. In this chapter, we'll explore Dart's syntax, key features, and why it's essential for Flutter development. Understanding Dart will help you write clean, efficient, and scalable Flutter applications.

2.3 WHY DART FOR FLUTTER?

Dart was chosen for Flutter for several reasons:

- **Fast Development:** Dart’s Just-in-Time (JIT) compilation supports Flutter’s hot reload feature, which allows instant UI updates during development without needing to restart the app.
- **High Performance:** Dart compiles to native ARM and x86 code for mobile devices using Ahead-of-Time (AOT) compilation, ensuring smooth performance on mobile and desktop platforms.
- **Consistent UI Rendering:** Dart enables Flutter’s unique UI rendering system, where widgets are drawn directly to the screen rather than relying on native components.
- **Easy to Learn:** Dart’s syntax is similar to languages like JavaScript, Java, and C#, making it approachable for developers coming from other object-oriented languages.

2.4 GETTING STARTED WITH DART

Before diving into Flutter development, let’s go over the basics of Dart. Here’s how you can start using Dart:

1. **DartPad**: The easiest way to experiment with Dart is through https://dartpad.dev/ an online editor where you can write and run Dart code in your browser.
2. **Installing Dart**: You can install Dart SDK locally on your system if you want to run Dart programs offline. However, if you’ve already installed Flutter, Dart comes bundled with it, so you don’t need to install it separately.

2.5 DART BASICS: SYNTAX AND STRUCTURE

Main Function

In Dart, every program starts with the main() function. It’s the entry point for all Dart applications, including Flutter apps.

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

Here, main() is the starting point, and print() is used to output text to the console.

Variables and Data Types

Dart is a statically-typed language, meaning variables must have a specific type. However, it also supports type inference, which means you don’t always have to explicitly declare the type.

- **Declaring Variables**:
  int age = 25;
  String name = 'John';
  bool isFlutterDev = true;
- **Using var for Type Inference**:
  var city = 'New York'; // Dart infers that city is of type String
  var number = 42;   // Dart infers that number is of type int
- **Null Safety**: Dart enforces null safety by default, meaning variables cannot hold null unless explicitly declared as nullable:
  int? maybeAge = null; // Using ? allows the variable to hold null

Basic Data Types

Dart has several core data types:

- **Numbers**: int for whole numbers and double for floating-point numbers.
  int quantity = 5;
  double price = 9.99;
- **Strings**: Strings in Dart are UTF-16 encoded and can be enclosed in either single or double quotes.
  String message = 'Hello, World!';
- **Booleans**: bool represents true or false values.
  bool isActive = true;
- **Lists (Arrays)**: Dart uses lists to store an ordered collection of elements.
  List<String> fruits = ['Apple', 'Banana', 'Mango'];
- **Maps (Dictionaries)**: A Map holds key-value pairs.
  Map<String, int> scores = {'John': 90, 'Alice': 85};

2.6 CONTROL FLOW STATEMENTS

Control flow statements in Dart include the usual suspects found in most programming languages: conditionals and loops.

Conditional Statements

- **If-Else Statement**:

```
int score = 85;
if (score >= 90) {
  print('Excellent');
} else if (score >= 70) {
  print('Good');
} else {
  print('Needs Improvement');
}
```

- **Switch Case**:

```
var grade = 'B';
switch (grade) {
  case 'A':
    print('Excellent');
    break;
  case 'B':
    print('Good');
    break;
  default:
    print('Needs Improvement');
}
```

Loops

- **For Loop**:

```
for (int i = 0; i < 5; i++) {
  print(i);
}
```

- **While Loop**:

```
int count = 0;
while (count < 5) {
  print(count);
  count++;
}
```

2.7 FUNCTIONS IN DART

Functions in Dart are used to organize and reuse code. Functions can return values or be void (returning no value).

Defining Functions

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

You can also use arrow syntax for short functions:

```
int multiply(int a, int b) => a * b;
```

Optional and Named Parameters: Dart allows functions to have optional or named parameters:

- **Optional Parameters**: Parameters can be made optional by enclosing them in square brackets [].

```
void greet(String name, [String title = '']) {
  print('Hello, $title $name');
}
```

- **Named Parameters**: Parameters can be made named by enclosing them in curly braces {}. This improves code readability when calling functions with multiple arguments.

```
void greet({required String firstName, String lastName = ''}) {
  print('Hello, $firstName $lastName');
}
```

2.8 OBJECT-ORIENTED PROGRAMMING (OOP) IN DART

Dart is an object-oriented language that supports classes, inheritance, and polymorphism.

Defining Classes

```
class Person {
  String name;
  int age;

  // Constructor
  Person(this.name, this.age);
  void greet() {
    print('Hello, my name is $name and I am $age years old.');
  }
}
```

Inheritance

Dart supports class inheritance, allowing you to reuse existing code.

```
class Student extends Person {
  String school;

  Student(String name, int age, this.school) : super(name, age);

  @override
  void greet() {
    print('Hello, I am $name, studying at $school.');
  }
}
```

2.9 ASYNCHRONOUS PROGRAMMING IN DART

Flutter apps often perform tasks that take time to complete, like fetching data from a server. Dart handles asynchronous programming using async and await.

Future

A Future represents a value that may be available in the future. You can handle it with then() or using async and await.

```
Future<String> fetchData() async {
  await Future.delayed(Duration(seconds: 2));
  return 'Data fetched!';
}


void main() async {
  print('Fetching...');
  var data = await fetchData();
  print(data);
}
```

2.10 DART IN FLUTTER

When you write Flutter apps, you're essentially writing Dart code. Key Flutter concepts like widgets, state, and layouts are built on Dart's object-oriented model. Understanding Dart makes you proficient in Flutter.

1. **Widgets**: Flutter's StatelessWidget and StatefulWidget classes are written in Dart. Understanding Dart classes and inheritance will help you build custom widgets.
2. **Asynchronous Programming**: Flutter often uses asynchronous functions, especially for tasks like fetching data from APIs or interacting with databases.
3. **Functional Programming**: Dart supports functional programming concepts, allowing Flutter apps to be reactive and declarative.

2.11 LET US SUM UP

In this chapter, we explored the essentials of Dart, the language behind Flutter. Dart's modern syntax, easy-to-learn structure, and powerful features like null safety, object-oriented programming, and asynchronous handling make it an ideal language for building mobile and web apps. As you dive deeper into Flutter, a solid understanding of Dart will enable you to write clean, efficient, and scalable code.

Check Your Progress-1

1. _____is an online editor where you can write and run Dart code in your browser

   A) NotePad B) DartPad C) WordPad D) NotePad++

2. When you write Flutter apps, you're essentially writing ______code.

   A) Java B) Cotlin C) Dart D) C

3. Dart handles asynchronous programming using async and await.

   A) async B) await C) Both A and B D) None of these

4. Dart is a general-purpose, object-oriented language that powers Flutter's ________ programming model.

   A) Reactive B) Active C) Passive D) Responsive

2.12 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1. B – DartPad
2. C – Dart
3. C – Both A and B
4. A - Reactive

2.13 FURTHER READING

- https://dartpad.dev/
- https://dart.dev/overview

2.14 ASSIGNMENT

1 What is dart?

2 What is DartPad?

3 Why dart is used for Flutter?

4 Explain control flow statements in Dart

5 Explain variable and data types in Dart

6 How can we define functions in Dart?

7 Explain asynchronous programming in Dart?

Unit-3: Getting Started with Flutter

3

Unit Structure

3.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Install and configure Flutter SDK on their system (Windows, macOS, or Linux) and set up development tools like Android Studio or Visual Studio Code.
- Create a new Flutter project from the command line and understand the basic structure of a Flutter project.
- Build and run a Flutter application on an emulator or physical device, and understand the difference between hot reload and hot restart.
- Modify the default Flutter project to display custom content and become familiar with writing and editing Dart code.
- Understand the role of widgets in Flutter and differentiate between stateless and stateful widgets.
- Leverage Flutter's hot reload feature to rapidly develop and debug Flutter applications without losing the application state.
- Understand the key files and directories in a Flutter project, including lib/main.dart, pubspec.yaml, and the platform-specific directories.

3.2 INTRODUCTION

Flutter has rapidly emerged as one of the most popular frameworks for building cross-platform applications with a single codebase. Whether you're targeting Android, iOS, web, or desktop, Flutter allows developers to build natively compiled apps with expressive and flexible UI. Its development speed, performance, and beautiful design make it a preferred choice for startups and large enterprises alike. In this chapter, we will walk through the steps of getting started with Flutter, from installation to writing your first app. We will learn Start building Flutter Android apps on Windows

3.3 H/W AND S/W REQUIREMENT TO INSTALL FLUTTER

To install and run Flutter, your Windows environment must meet the following hardware and software requirements.

Hardware requirements

Your Windows Flutter development environment must meet the following minimal hardware requirements.

Requirement	Minimum	Recommended
x86_64 CPU Cores	4	8
Memory in GB	8	16
Display resolution in pixels	WXGA (1366 x 768)	FHD (1920 x 1080)
Free disk space in GB	11.0	60.0

Software requirements

To write and compile Flutter code for Android, you must have the following version of Windows and the listed software packages.

Operating system

Flutter supports 64-bit version of Microsoft Windows 10 or later. These versions of Windows should include the required Windows PowerShell 5 or later.

Development tools

Download and install the Windows version of the following packages:

- Git for Windows 2.27 or later to manage source code.
- Android Studio 2023.3.1 (Jellyfish) or later to debug and compile Java or Kotlin code for Android. Flutter requires the full version of Android Studio.

When you run the current version of flutter doctor, it might list a different version of one of these packages. If it does, install the version it recommends.

Configure a text editor or IDE

You can build apps with Flutter using any text editor or integrated development environment (IDE) combined with Flutter's command-line tools.

Using an IDE with a Flutter extension or plugin provides code completion, syntax highlighting, widget editing assists, debugging, and other features.

Popular options include:

- **Visual Studio Code 1.77** or later with the Flutter extension for VS Code.
- Android Studio 2023.3.1 (Jellyfish) or later with the Flutter plugin for IntelliJ.
- IntelliJ IDEA 2023.3 or later with the Flutter plugin for IntelliJ.

The Flutter team recommends installing Visual Studio Code 1.77 or later and the Flutter extension for VS Code. This combination simplifies installing the Flutter SDK.

3.4 INSTALL THE FLUTTER SDK

Before we start building with Flutter, we need to set up the development environment. The steps below guide you through the installation process for different platforms: Windows, macOS, and Linux.

To install the Flutter SDK, you can use the VS Code Flutter extension or download and install the Flutter bundle yourself.

To install Flutter, download the Flutter SDK bundle from its archive, move the bundle to where you want it stored, then extract the SDK.

1. Download the following installation bundle to get the latest stable release of the Flutter SDK.

   https://storage.googleapis.com/flutter_infra_release/releases/stable/windows/flutter_windows_3.24.3-stable.zip

   The Flutter SDK should download to the download folder.
2. Create a folder where you can install Flutter.

   Consider creating a directory at %USERPROFILE% (C:\Users\{username}) or %LOCALAPPDATA% (C:\Users\{username}\AppData\Local).
3. Extract the file into the directory you want to store the Flutter SDK.

   ```
   PS C:\> Expand-Archive `
       -Path $env:USERPROFILE\Downloads\flutter_windows_3.24.3-stable.zip `
       -Destination $env:USERPROFILE\dev\
   ```

   When finished, the Flutter SDK should be in the C:\user\{username}\dev\flutter directory.

3.5 UPDATE YOUR WINDOWS PATH VARIABLE

To run Flutter commands in PowerShell, add Flutter to the PATH environment variable. This section presumes that you installed the Flutter SDK in %USERPROFILE%\dev\flutter.

1. Press Windows + Pause.

   If your keyboard lacks a Pause key, try Windows + Fn + B.

   The **System > About** dialog displays.
2. Click **Advanced System Settings** > **Advanced** > **Environment Variables...**

The **Environment Variables** dialog displays.

3. In the **User variables for (username)** section, look for the **Path** entry.
    1. If the entry exists, double-click on it.

    The **Edit Environment Variable** dialog displays.

        1. Double-click in an empty row.
        2. Type %USERPROFILE%\dev\flutter\bin.
        3. Click the **%USERPROFILE%\dev\flutter\bin** entry.
        4. Click **Move Up** until the Flutter entry sits at the top of the list.
        5. Click **OK** three times.

    2. If the entry doesn't exist, click **New...**.

    The **Edit Environment Variable** dialog displays.

        1. In the **Variable Name** box, type Path.
        2. In the **Variable Value** box, type %USERPROFILE%\dev\flutter\bin
        3. Click **OK** three times.

4. To enable these changes, close and reopen any existing command prompts and PowerShell instances.

3.6 CONFIGURE ANDROID DEVELOPMENT

1. Launch **Android Studio**.

   The **Welcome to Android Studio** dialog displays.
2. Follow the **Android Studio Setup Wizard**.
3. Install the following components:
    - **Android SDK Platform, API 35.0.1**
    - **Android SDK Command-line Tools**
    - **Android SDK Build-Tools**
    - **Android SDK Platform-Tools**
    - **Android Emulator**

3.7 CONFIGURE YOUR TARGET ANDROID DEVICE

Set up the Android emulator

To configure your Flutter app to run in an Android emulator, follow these steps to create and select an emulator.

1. Enable VM acceleration on your development computer.
2. Start **Android Studio**.
3. Go to the **Settings** dialog to view the **SDK Manager**.
    1. If you have a project open, go to **Tools** > **Device Manager**.
    2. If the **Welcome to Android Studio** dialog displays, click the **More Options** icon that follows the **Open** button and click **Device Manager** from the dropdown menu.
4. Click **Virtual**.
5. Click **Create Device**.

The **Virtual Device Configuration** dialog displays.

6. Select either **Phone** or **Tablet** under **Category**.
7. Select a device definition. You can browse or search for the device.
8. Click **Next**.
9. Click **x86 Images**.
10. Click one system image for the Android version you want to emulate.
    1. If the desired image has a **Download** icon to the right of the **Release Name**, click it.

       The **SDK Quickfix Installation** dialog displays with a completion meter.
    2. When the download completes, click **Finish**.
11. Click **Next**.

    The **Virtual Device Configuration** displays its **Verify Configuration** step.
12. To rename the Android Virtual Device (AVD), change the value in the **AVD Name** box.
13. Click **Show Advanced Settings** and scroll to **Emulated Performance**.
14. From the **Graphics** dropdown menu, select **Hardware - GLES 2.0**.

    This enables hardware acceleration and improves rendering performance.
15. Verify your AVD configuration. If it is correct, click **Finish**.
16. In the **Device Manager** dialog, click the **Run** icon to the right of your desired AVD. The emulator starts up and displays the default canvas for your selected Android OS version and device.

3.8 AGREE TO ANDROID LICENSES

Before you can use Flutter and after you install all prerequisites, agree to the licenses of the Android SDK platform.

1. Open an elevated console window.
2. Run the following command to enable signing licenses.

   C:> flutter doctor --android-licenses

   If you accepted the Android Studio licenses at another time, this command returns:

   ```
   [=======================================] 100% Computing updates...
   All SDK package licenses accepted.
   ```

   You can skip the next step.
3. Before agreeing to the terms of each license, read each with care.

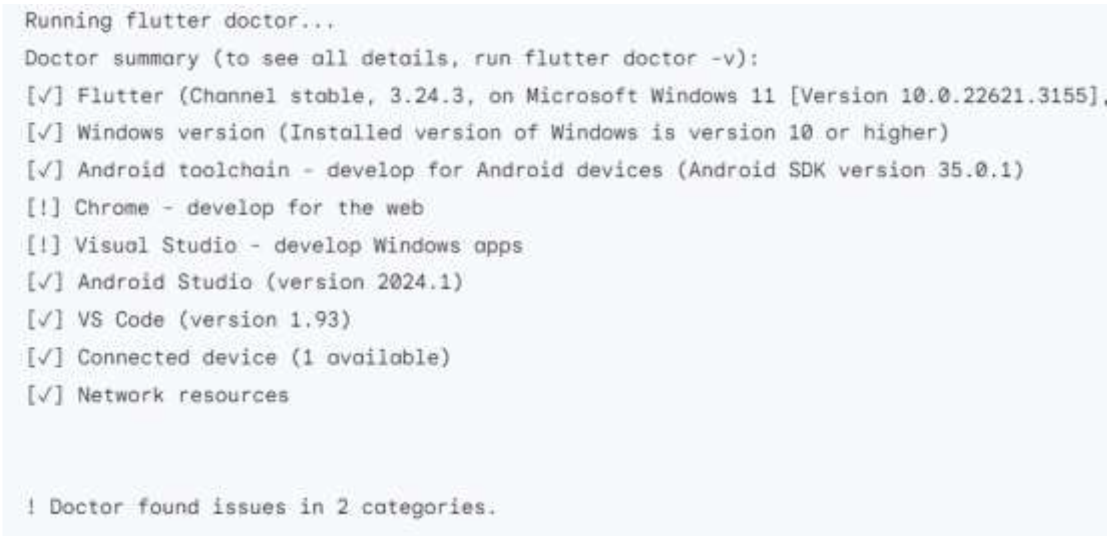
3.9 CHECK YOUR DEVELOPMENT SETUP

The flutter doctor command validates that all components of a complete Flutter development environment for Windows.

1. Open PowerShell.
2. To verify your installation of all the components, run the following command.

   PS C:> flutter doctor

As you chose to develop for Android, you do not need *all* components. If you followed this guide, the result of your command should resemble:



```
Running flutter doctor...
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, 3.24.3, on Microsoft Windows 11 [Version 10.0.22621.3155],
[√] Windows version (Installed version of Windows is version 10 or higher)
[√] Android toolchain - develop for Android devices (Android SDK version 35.0.1)
[!] Chrome - develop for the web
[!] Visual Studio - develop Windows apps
[√] Android Studio (version 2024.1)
[√] VS Code (version 1.93)
[√] Connected device (1 available)
[√] Network resources


! Doctor found issues in 2 categories.
```

Troubleshoot Flutter doctor issues

When the flutter doctor command returns an error, it could be for Flutter, VS Code, Android Studio, the connected device, or network resources.

If the flutter doctor command returns an error for any of these components, run it again with the verbose flag.

PS C:> flutter doctor -v

Check the output for other software you might need to install or further tasks to perform. If you change the configuration of your Flutter SDK or its related components, run flutter doctor again to verify the installation.

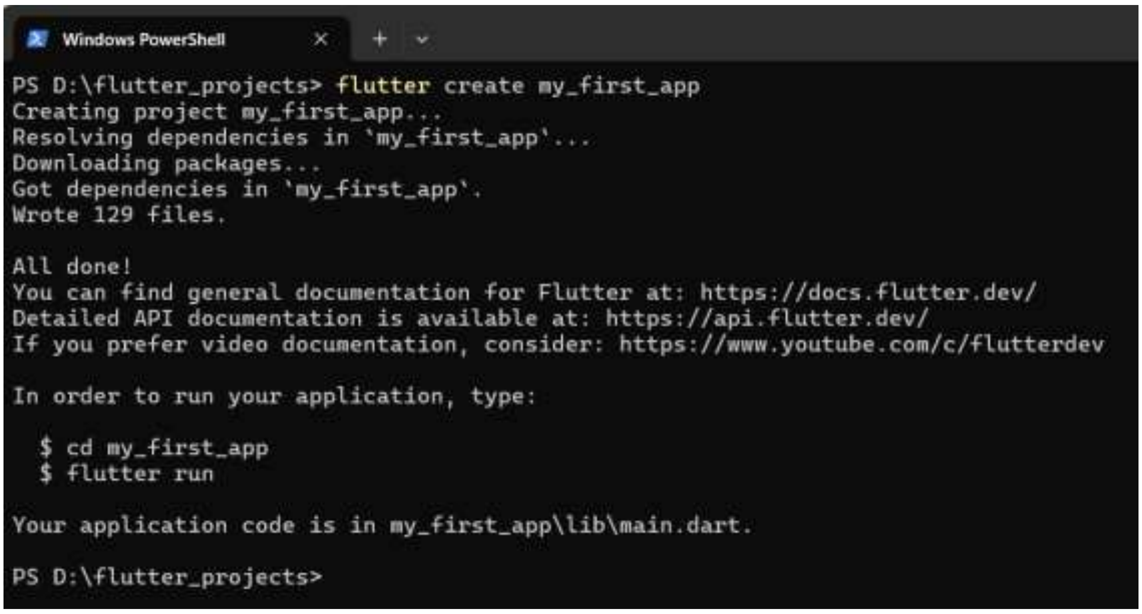
Congratulations. Having installed all prerequisites and the Flutter SDK, you can start developing Flutter apps for Android on Windows. Now you can start developing Android on Windows apps with Flutter

3.10 CREATING YOUR FIRST FLUTTER APP

Now that the environment is set up, let's create and run your first Flutter app.

Create a New Flutter Project

1. Open your terminal and navigate to the directory where you want to create your project.
2. Run the **flutter create my_first_app** command to create a new Flutter project as shown below:



```
PS D:\flutter_projects> flutter create my_first_app
Creating project my_first_app...
Resolving dependencies in `my_first_app`...
Downloading packages...
Got dependencies in `my_first_app`.
Wrote 129 files.

All done!
You can find general documentation for Flutter at: https://docs.flutter.dev/
Detailed API documentation is available at: https://api.flutter.dev/
If you prefer video documentation, consider: https://www.youtube.com/c/flutterdev

In order to run your application, type:

  $ cd my_first_app
  $ flutter run

Your application code is in my_first_app\lib\main.dart.

PS D:\flutter_projects>
```

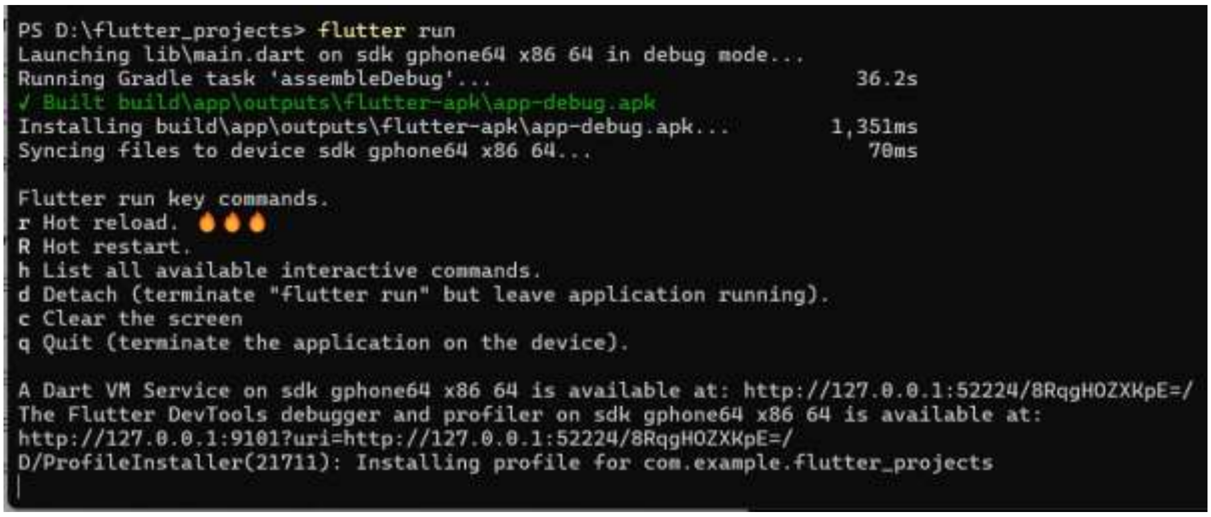
This command will create a new Flutter project with the necessary files and directories.

3. Navigate to your project directory:

   cd my_first_app

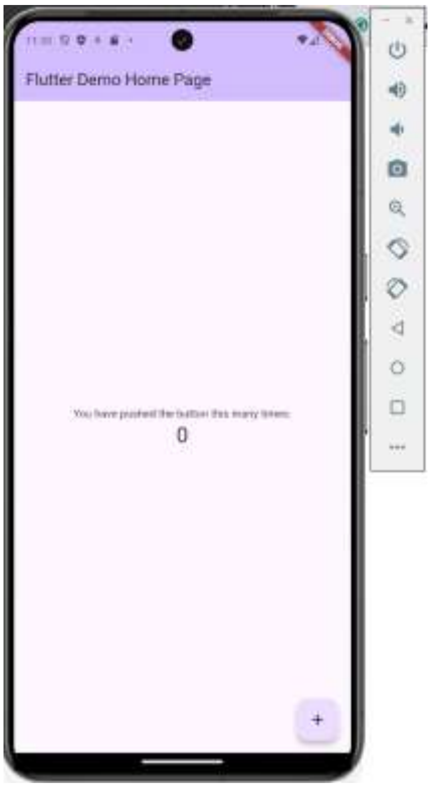
Run the App

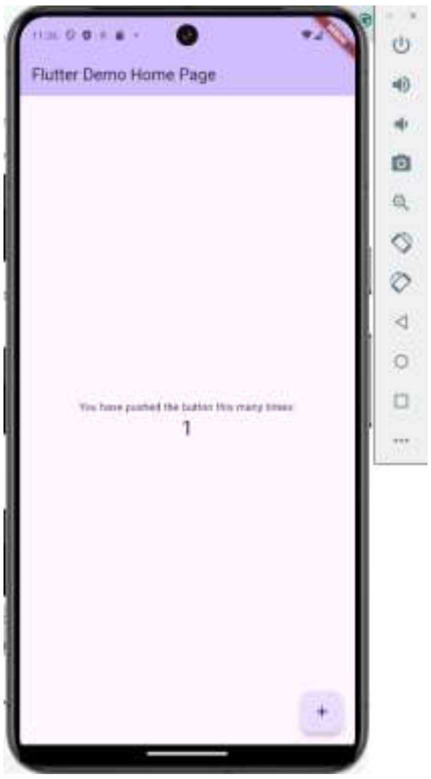
1. Run the following command to launch the app on the available emulator or device:



```
PS D:\flutter_projects> flutter run
Launching lib\main.dart on sdk gphone64 x86 64 in debug mode...
Running Gradle task 'assembleDebug'...                              36.2s
√ Built build\app\outputs\flutter-apk\app-debug.apk
Installing build\app\outputs\flutter-apk\app-debug.apk...         1,351ms
Syncing files to device sdk gphone64 x86 64...                       70ms

Flutter run key commands.
r Hot reload. 🔥🔥🔥
R Hot restart.
h List all available interactive commands.
d Detach (terminate "flutter run" but leave application running).
c Clear the screen
q Quit (terminate the application on the device).

A Dart VM Service on sdk gphone64 x86 64 is available at: http://127.0.0.1:52224/8RqgHOZXKpE=/
The Flutter DevTools debugger and profiler on sdk gphone64 x86 64 is available at:
http://127.0.0.1:9101?uri=http://127.0.0.1:52224/8RqgHOZXKpE=/
D/ProfileInstaller(21711): Installing profile for com.example.flutter_projects
|
```

If you have an Android or iOS emulator set up, you should see the default Flutter app running on your device. The app will display a counter that increments each time you press the "plus" button.





Understanding the Project Structure

- **lib/main.dart**: This is the main entry point of your Flutter app. The main.dart file contains the main() function, which is the starting point of any Dart application. By default, it also contains a StatefulWidget that displays a simple counter app.
- **pubspec.yaml**: This file is the configuration file for your Flutter project. It includes information about your app's dependencies, such as plugins, fonts, and assets.
- **android/ and ios/ folders**: These folders contain platform-specific code for Android and iOS. As a Flutter developer, you typically won't need to modify these files unless you need custom native functionality.

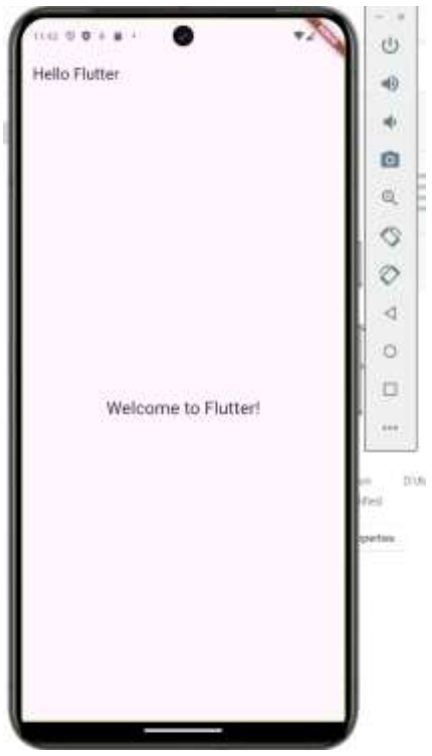
3.11 WRITING YOUR FIRST FLUTTER CODE

Let's modify the default Flutter app to display a custom message. Follow the steps below to change the basic counter app into a simple "Hello, Flutter!" app.

1. Open the lib/main.dart file in your IDE.
2. Replace the default code with the following:

```
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('Hello Flutter'),
        ),
        body: Center(
          child: Text(
            'Welcome to Flutter!',
            style: TextStyle(fontSize: 24),
          ),
        ),
      ),
    );
  }
}
```

Save the file, and if you're using hot reload, you should see the changes applied immediately. The app will now display a "Welcome to Flutter!" message at the center of the screen.



Now you should understand that the lib/main.dart is the main entry point of your Flutter app and it contains the main() function, which is the starting point of any Dart application. By default, it also contains a StatefulWidget that displays a simple counter app that can be modified as per requirement.

3.12 KEY CONCEPTS

Widgets: In Flutter, everything is a widget. Widgets are the building blocks of the UI.

There are two types of widgets in Flutter:

- **Stateless Widgets**: Widgets that do not change over time, such as static text or icons.
- **Stateful Widgets**: Widgets that can change based on user interaction or other factors, like forms or dynamic content.

Hot Reload vs. Hot Restart

- **Hot Reload**: Allows you to instantly view the changes you've made in the app without losing the current app state.
- **Hot Restart**: Completely rebuilds the app and clears the app's state.

3.13 LET US SUM UP

In this chapter, we covered the basics of getting started with Flutter. You learned how to install Flutter, create your first project, and run a simple app. We also touched on the project structure and how to write basic Flutter code. In the upcoming chapters, we will dive deeper into Flutter's widgets, navigation, state management, and more.

3.14 FURTHER READING

- https://docs.flutter.dev/get-started/install/windows/mobile

3.15 ASSIGNMENT

1. Write steps to install flutter?
2. Write hardware and software requirement to install Flutter
3. Define: Widget, Stateful Widgets and Stateless Widgets
4. What is difference between Hot Reload and Hot Restart?
5. Explain Flutter project structure

Block-2

Flutter Basics

Unit-4: Understanding Flutter Widgets

4

Unit Structure

4.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Explain what widgets are and how they form the core of Flutter’s UI design system.
- Identify the key differences between stateless and stateful widgets and understand when to use each.
- Utilize essential widgets like Text, Container, Row, Column, Image, and Button to build simple and complex UI elements.
- Combine and nest multiple widgets within each other to create structured, layered user interfaces.
- Use layout widgets such as Align, Padding, Expanded, and Flexible to control the positioning and spacing of UI elements.
- Create custom stateless and stateful widgets to encapsulate reusable functionality and improve code organization.
- Understand the hierarchy of the widget tree and how Flutter uses it to render the UI.
- Manage state within stateful widgets to create responsive and dynamic user interfaces that react to user interaction.

4.2 INTRODUCTION

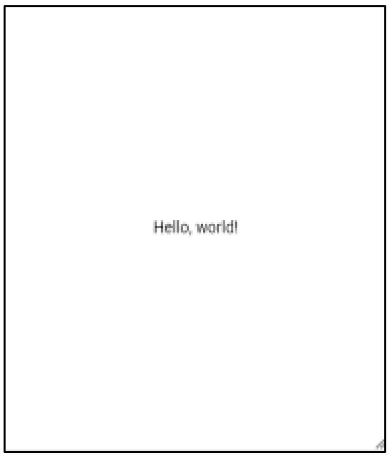
Flutter’s core building blocks are widgets. In fact, everything in Flutter is a widget—from layout elements like buttons and text to the structure and behavior of your app. Widgets define not only the UI but also how it reacts to user interactions and changes in state. This chapter introduces the concept of widgets in Flutter, explains the different types of widgets, and demonstrates how to compose them to build beautiful, responsive UIs.

4.3 WHAT ARE WIDGETS?

Widgets are the fundamental building blocks in Flutter. Everything you see on the screen, such as text, images, buttons, and layouts, is a widget. Even invisible elements that control layout, such as padding and alignment, are widgets. Flutter’s widget system is highly flexible, allowing developers to create complex UIs by combining simple widgets in different ways.

The minimal Flutter app simply calls the runApp() function with a widget:

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

void main() {
  runApp(
    const Center(
      child: Text(
        'Hello, world!',
        textDirection: TextDirection.ltr,
      ),
    ),
  );
}
```


Hello, world!

Figute 4.1: Simple Widget

The runApp() function takes the given Widget and makes it the root of the widget tree. In this example, the widget tree consists of two widgets, the Center widget and its child, the Text widget. The framework forces the root widget to cover the screen, which means the text "Hello, world" ends up centered on screen. The text direction needs to be specified in this instance; when the MaterialApp widget is used, this is taken care of for you, as demonstrated later.

When writing an app, you'll commonly author new widgets that are subclasses of either StatelessWidget or StatefulWidget, depending on whether your widget manages any state. A widget's main job is to implement a build() function, which describes the widget in terms of other, lower-level widgets. The framework builds those widgets in turn until the process bottoms out in widgets that represent the underlying RenderObject, which computes and describes the geometry of the widget.

Widget Tree

Every Flutter app consists of a widget tree. The root of this tree is the runApp() function, which takes the widget that forms the app's root and builds a tree of nested widgets beneath it.

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

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

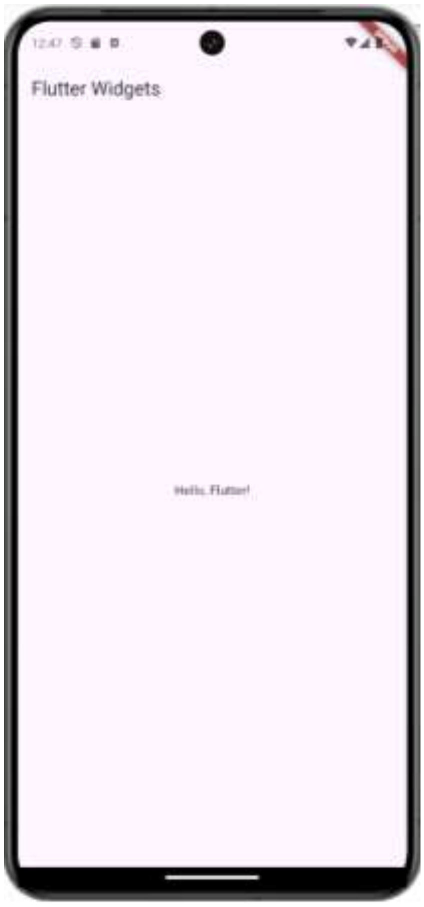


Figure 4.2: Widget Tree

In this example, the widget tree consists of a MaterialApp, a Scaffold, an AppBar, and a Text widget. Each widget in the tree is responsible for rendering a part of the UI.

4.4 TYPES OF WIDGETS

Widgets in Flutter can be categorized into two main types: **Stateless Widgets** and **Stateful Widgets**.

Stateless Widgets: A **StatelessWidget** is a widget that does not change its state during its lifetime. It renders static content that remains the same unless the entire widget is rebuilt.

Example:

```
import 'package:flutter/material.dart';
void main() {
  runApp(const MyApp());
}

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return const MaterialApp(
      home: Scaffold(
        body: MyStatelessWidget(),
      ),
    );
  }
}
class MyStatelessWidget extends
StatelessWidget {
  const MyStatelessWidget({super.key});

  @override
  Widget build(BuildContext context) {
    return Center(child: const Text('I am a
Stateless Widget'));
  }
}
```

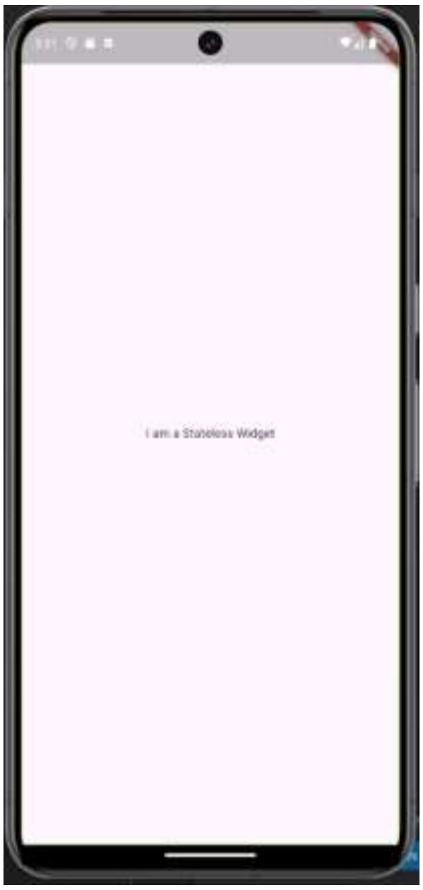

I am a Stateless Widget

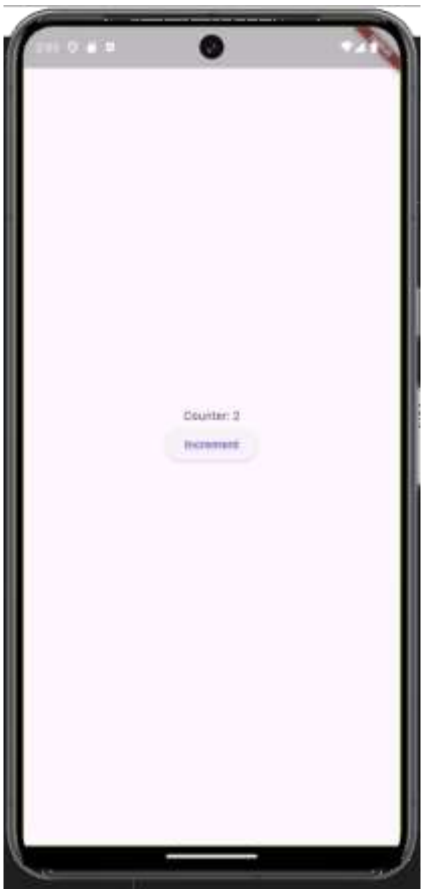
Figure 4.3 StatelessWidget

Stateless widgets are ideal for displaying static UI components like labels, icons, and buttons that don’t change dynamically as shown in figure 4.3.

Stateful Widgets: A **StatefulWidget** is a widget that can change its state during its lifetime. The state of the widget can be updated based on user interaction or other events.

Example:

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

void main() {
  runApp(
    const MaterialApp(
      home: Scaffold(
        body: Center(
          child: MyStatefulWidget(),
        ),
      ),
    ),
  );
}

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


class _MyStatefulWidgetState extends State<MyStatefulWidget> {
  int _counter = 0;
  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }
  @override
  Widget build(BuildContext context) {
    return Column(
mainAxisAlignment: MainAxisAlignment.center,
      children: [
        Text('Counter: $_counter'),
        ElevatedButton(
          onPressed: _incrementCounter,
          child: const Text('Increment'),
        ),
      ],
    );
  }
}
```



Figute 4.4 Stateful Widgets

Here, MyStatefulWidget has a counter that changes when the button is pressed. The setState() method is used to notify Flutter that the state has changed and the UI should be updated.

4.5 COMMON FLUTTER WIDGETS

Common Flutter Widgets

Flutter provides a wide variety of widgets to build both simple and complex user interfaces. Below are some of the most commonly used widgets:

Text Widget

The Text widget is used to display a string of text in the app.

```
Text('Hello, Flutter!',
  style: TextStyle(fontSize: 24, fontWeight: FontWeight.bold),
);
```

Container Widget

The Container widget is a flexible widget that can be used to add padding, margins, borders, or backgrounds to child widgets.

```
Container(
  padding: EdgeInsets.all(16.0),
  margin: EdgeInsets.all(8.0),
  decoration: BoxDecoration(
    color: Colors.blue,
    borderRadius: BorderRadius.circular(10),
  ),
  child: Text('I am inside a container'),
);
```

Row and Column Widgets

The Row and Column widgets allow you to arrange child widgets in a horizontal or vertical direction, respectively.

Row:

```
Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: [
    Icon(Icons.star),
    Text('Starred'),
  ],
);
```

Column:

```
Column(
  crossAxisAlignment: CrossAxisAlignment.start,
  children: [
    Text('Item 1'),
    Text('Item 2'),
    Text('Item 3'),
  ],
);
```

Image Widget

The Image widget is used to display images in your Flutter app. You can load images from assets, network, or files.

```
Image.network('https://example.com/image.png');
```

Button Widgets

Flutter offers several button widgets, such as ElevatedButton, TextButton, and IconButton.

Example of an ElevatedButton:

```
ElevatedButton(
  onPressed: () {
    print('Button Pressed');
  },
  child: Text('Press Me'),
);
```

ListView Widget: The ListView widget is used to display a scrollable list of items.

```
ListView(
  children: [
    ListTile(title: Text('Item 1')),
    ListTile(title: Text('Item 2')),
    ListTile(title: Text('Item 3')),
  ],
);
```

4.6 WIDGET COMPOSITION AND NESTING

One of Flutter's greatest strengths is the ability to compose widgets by nesting them inside one another. This allows you to create complex UIs from smaller, reusable components.

For example, a button with a label inside a container:

```
Container(
  padding: EdgeInsets.all(16),
  decoration: BoxDecoration(
    color: Colors.blueAccent,
    borderRadius: BorderRadius.circular(8),
  ),
  child: ElevatedButton(
    onPressed: () {},
    child: Text('Click Me'),
  ),
);
```

In this case, we nest an ElevatedButton inside a Container, giving it padding and rounded corners.

4.7 LAYOUT IN FLUTTER

Flutter provides a flexible layout system that allows you to position widgets within your app in various ways.

Alignment and Padding

Align Widget: Controls the alignment of a widget within its parent.

```
Align(
  alignment: Alignment.center,
  child: Text('Centered Text'),
);
```

Padding Widget: Adds space around a widget.

```
Padding(
  padding: EdgeInsets.all(16.0),
  child: Text('Text with Padding'),
);
```

Expanded and Flexible Widgets

The Expanded and Flexible widgets are used to control how widgets grow and shrink inside a Row or Column.

Expanded: Expands a child widget to fill the available space.

```
Row(
  children: [
    Expanded(child: Text('Item 1')),
    Expanded(child: Text('Item 2')),
  ],
);
```

Flexible: Flexibly sizes a widget, but allows it to take up only as much space as needed.

```
Flexible (
  child: Text ('Flexible Widget'),
);
```

4.8 BUILDING CUSTOM WIDGETS

One of the key features of Flutter is the ability to create custom widgets. Custom widgets allow you to encapsulate code that is reused across the app, improving maintainability and readability.In section 4.3 we have already learnt how to create a Custom Stateless Widget and a Custom Stateful Widget

Check Your Progress-1

a) One of Flutter's greatest strengths is the ability to compose widgets by nesting them inside one another. (True/False)

b) Widgets in Flutter can be categorized into _________main types

c) Views are the fundamental building blocks in Flutter. (True/False)

d) The _______ widget is a flexible widget that can be used to add padding, margins, borders, or backgrounds to child widgets.

e) The minimal Flutter app simply calls the runApp() function with a widget (True/False)

4.9 LET US SUM UP

In this chapter, we covered the basics of getting started with Flutter. You learned how to install Flutter, create your first project, and run a simple app. We also touched on the project structure and how to write basic Flutter code. In the upcoming chapters, we will dive deeper into Flutter’s widgets, navigation, state management, and more.

4.10 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a) True 1-b) Two 1-c) Fakse 1-d) Container 1-e) True

4.11 FURTHER READING

- https://docs.flutter.dev/ui
- https://docs.flutter.dev/ui/widgets

4.12 ASSIGNMENT

1. What is layout?
2. Explain alignment and padding
3. What is widget composition and nesting?
4. Explain common flutter widgets with code snippet and use.

Unit-5: Building User Interfaces with Flutter

5

Unit Structure

5.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Comprehend the fundamental principles of Flutter’s widget-based architecture and how it underpins the construction of user interfaces.
- Identify and implement the appropriate use of stateless and stateful widgets for different UI components and app functionalities.
- Effectively use common Flutter widgets such as Container, Text, Row, Column, ListView, and Stack to build various UI elements.
- Use layout widgets like Row, Column, Expanded, Flexible, and Stack to create responsive and adaptive UI designs.
- Build UIs that respond dynamically to different screen sizes, orientations, and constraints using MediaQuery, LayoutBuilder, and OrientationBuilder.
- Apply custom themes using ThemeData and customize individual widgets to create consistent, attractive, and branded user interfaces.
- Understand how to manage UI state in Flutter applications, ensuring that the UI reacts to user interactions and changes.
- Add both implicit and explicit animations to Flutter applications, using widgets like AnimatedContainer and AnimationController to enhance user experience.
- Design and implement reusable custom widgets to simplify UI development and maintain clean code architecture.
- Understand best practices for ensuring that Flutter apps run smoothly and efficiently, avoiding unnecessary rebuilds and performance bottlenecks in the UI.

5.2 INTRODUCTION

In the fast-paced world of app development, creating efficient, visually appealing, and user-friendly interfaces is crucial for user engagement and retention. Flutter, an open-source UI software development kit (SDK) developed by Google, has revolutionized the way developers build natively compiled applications for mobile, web, and desktop from a single codebase. This chapter focuses on the practical aspects of building user interfaces (UI) using Flutter, providing insights into its flexible and powerful tools for UI creation.

We will dive deep into the fundamental components of Flutter’s UI toolkit, including widgets, layouts, themes, and state management. By the end of this chapter, you should have a strong foundation for designing responsive, beautiful, and functional interfaces using Flutter.

5.3 WHAT MAKES FLUTTER A STRONG CHOICE FOR UI DEVELOPMENT?

Flutter has grown in popularity because of its unique features that simplify UI development. Here’s why Flutter stands out:

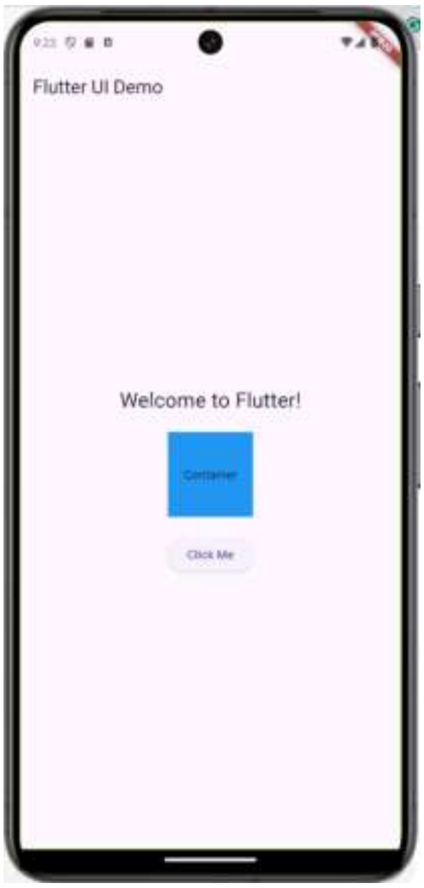
- **Single Codebase for Multiple Platforms:** With Flutter, developers can create apps for Android, iOS, Web, and even desktop using a single codebase.
- **Hot Reload:** One of Flutter's most loved features, it allows developers to instantly see the changes they make to the UI without restarting the app. This dramatically reduces development time.
- **Customizable Widgets:** Flutter provides a vast collection of pre-built widgets that can be customized to build native-looking UIs for both iOS and Android.
- **Performance:** Since Flutter compiles to native ARM code, its performance is comparable to native applications.
- **Expressive and Flexible UI:** Flutter’s widget-based architecture and layered design enable fine-grained control over every pixel, making it possible to create highly interactive and visually impressive UIs.

5.4 UNDERSTANDING THE FLUTTER UI STRUCTURE

In Flutter, everything is a widget. From the smallest UI components like buttons and text to more complex components like layouts and navigation controls, widgets form the building blocks of every Flutter application. Widgets in Flutter are classified into two categories: **Stateless** and **Stateful**.

Let’s build a simple UI with a text widget, a button, and a container to demonstrate basic layout widgets.

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

void main() {
  runApp(const MyHomePage());
}

class MyHomePage extends StatelessWidget {
  const MyHomePage({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
        home: Scaffold(
        appBar: AppBar(
        title: const Text('Flutter UI Demo'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: <Widget>[
            const Text(
              'Welcome to Flutter!',
              style: TextStyle(fontSize: 24),
            ),
            const SizedBox(height: 20),
            Container(
              height: 100,
              width: 100,
              color: Colors.blue,
              child: const Center(child: Text('Container')),
            ),
            const SizedBox(height: 20),
            ElevatedButton(
              onPressed: () {},
              child: const Text('Click Me'),
            ),
          ],
        ),
      ),
    ));
  }
}
```



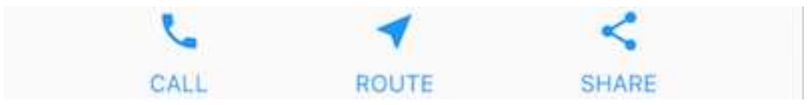
In this example:

- **Scaffold** provides the basic layout structure including an app bar.
- **Column** vertically arranges the Text, Container, and Button.
- **SizedBox** is used to add spacing between widgets.

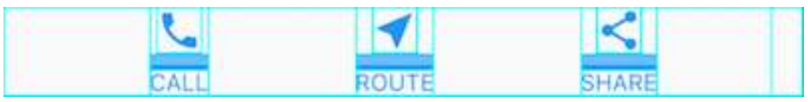
5.5 KEY LAYOUT CONCEPTS

The core of Flutter's layout mechanism is widgets. In Flutter, almost everything is a widget—even layout models are widgets. The images, icons, and text that you see in a Flutter app are all widgets. But things you don't see are also widgets, such as the rows, columns, and grids that arrange, constrain, and align the visible widgets.

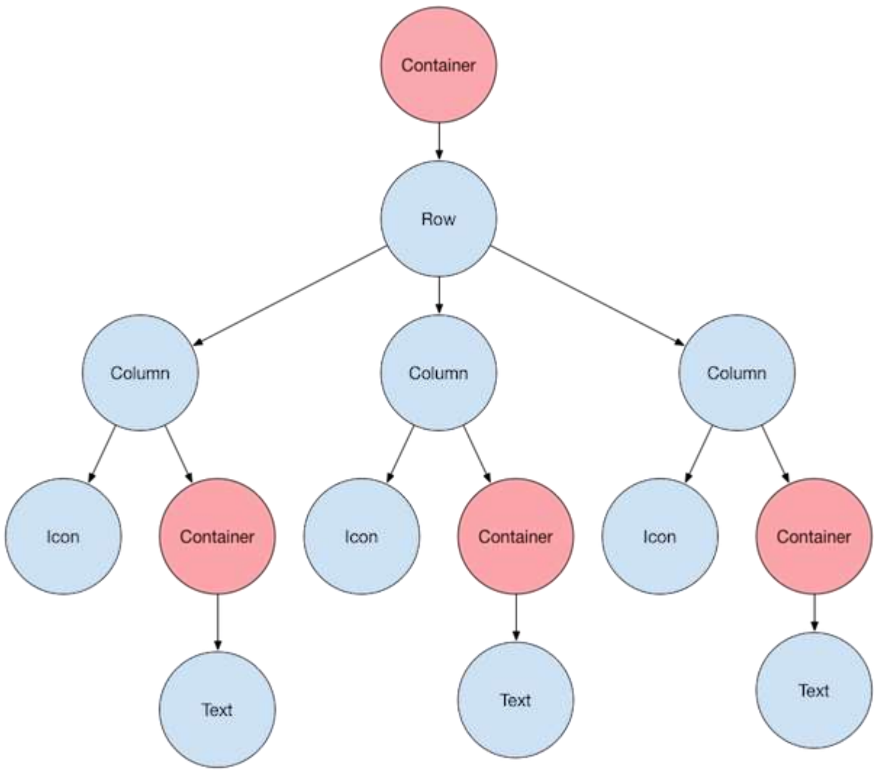
You create a layout by composing widgets to build more complex widgets. For example, the below screenshot below shows 3 icons with a label under each one:



The below screenshot displays the visual layout, showing a row of 3 columns where each column contains an icon and a label.



Here's a diagram of the widget tree for this UI:



Most of this should look as you might expect, but you might be wondering about the containers (shown in pink). Container is a widget class that allows you to customize its child widget. Use a Container when you want to add padding, margins, borders, or background color, to name some of its capabilities.

In this example, each Text widget is placed in a Container to add margins. The entire Row is also placed in a Container to add padding around the row.

The rest of the UI in this example is controlled by properties. Set an Icon's color using its color property. Use the Text.style property to set the font, its color, weight, and so on. Columns and rows have properties that allow you to specify how their children are aligned vertically or horizontally, and how much space the children should occupy.

Flutter uses a flexible layout system that allows developers to create responsive UIs. Let’s take a closer look at some of the key layout concepts.

MainAxisAlignment and CrossAxisAlignment

Both Row and Column have properties called mainAxisAlignment and crossAxisAlignment that control how their children are aligned within the available space.

- **MainAxisAlignment** aligns widgets along the main axis (horizontal for Row and vertical for Column).
- **CrossAxisAlignment** aligns widgets along the cross-axis (vertical for Row and horizontal for Column).

```
Row(
  mainAxisAlignment: MainAxisAlignment.center,
  crossAxisAlignment: CrossAxisAlignment.start,
  children: <Widget>[
    Icon(Icons.star),
    Icon(Icons.favorite),
    Icon(Icons.share),
  ],
)
```

In the above example, the icons are aligned horizontally in the center, and the cross-axis alignment is set to start, meaning the icons will be aligned at the top edge of the row.

Flexible and Expanded

When building UIs that need to stretch or fill available space, Flexible and Expanded widgets come in handy.

- **Expanded**: Forces a child widget to fill the available space.
- **Flexible**: Allows the child widget to take up a flexible amount of space within a Row or Column.

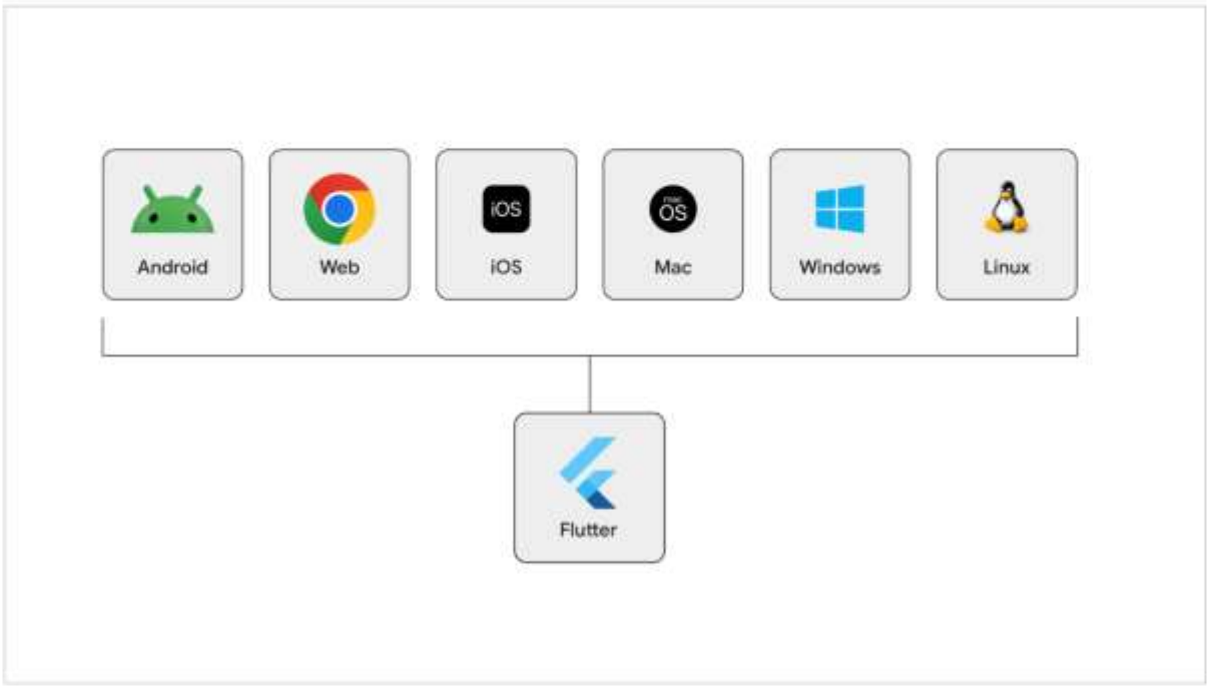
```
Row(
  children: <Widget>[
    Flexible(
      child: Container(color: Colors.red, height: 100),
    ),
    Flexible(
      child: Container(color: Colors.green, height: 100),
    ),
    Expanded(
      child: Container(color: Colors.blue, height: 100),
    ),
  ],
)
```

5.6 RESPONSIVE LAYOUTS

One of Flutter's primary goals is to create a framework that allows you to develop apps from a single codebase that look and feel great on any platform.

This means that your app might appear on screens of many different sizes, from a watch, to a foldable phone with two screens, to a high-definition monitor. And your input device might be a physical or virtual keyboard, a mouse, a touchscreen, or any number of other devices.

Two terms that describe these design concepts are *adaptive* and *responsive*. Ideally, you'd want your app to be *both* but what, exactly, does this mean?



What is responsive vs adaptive?

An easy way to think about it is that responsive design is about fitting the UI *into* the space and adaptive design is about the UI being *usable* in the space.

So, a responsive app adjusts the placement of design elements to *fit* the available space. And an adaptive app selects the appropriate layout and input devices to be usable *in* the available space. For example, should a tablet UI use bottom navigation or side-panel navigation?

Responsive design is essential for ensuring that your app looks and functions well on various screen sizes and device types. Flutter provides several powerful layout widgets that allow developers to create flexible and adaptive user interfaces.

Building UIs that look good across different screen sizes is essential. Flutter offers several approaches to build responsive designs.

Container: The Container widget is one of the most commonly used layout elements in Flutter. It allows you to wrap other widgets and control their size, padding, margins, alignment, and background decoration.
Container(

```
  height: 200,
  width: 200,
  padding: EdgeInsets.all(10),
  margin: EdgeInsets.all(10),
  decoration: BoxDecoration(
    color: Colors.blue,
    borderRadius: BorderRadius.circular(10),
  ),
  child: Text('I am inside a container'),
);
```

Stack: The Stack widget allows for layering of widgets on top of each other. This is useful for building more complex layouts where elements need to overlap, such as floating buttons or notifications.

```
Stack(
  children: [
    Container(
      height: 200,
      width: 200,
      color: Colors.red,
    ),
    Positioned(
      top: 50,
      left: 50,
      child: Container(
        height: 100,
        width: 100,
        color: Colors.green,
      ),
    ),
  ],
);
```

Flex and Expanded: Flex widgets, such as Row and Column, allow you to arrange children horizontally or vertically. The Expanded and Flexible widgets inside a Flex parent allow child widgets to grow and fill available space proportionally.

```
Row(
  children: [
    Expanded(child: Container(color: Colors.blue, height: 100)),
    Expanded(child: Container(color: Colors.red, height: 100)),
    Expanded(child: Container(color: Colors.green, height: 100)),
  ],
);
```

MediaQuery: MediaQuery provides information about the size of the screen and its characteristics. You can use it to create adaptive layouts that respond to changes in screen size.

```
@override
Widget build(BuildContext context) {
  var screenSize = MediaQuery.of(context).size;

  return Container(
    width: screenSize.width * 0.5,
    height: screenSize.height * 0.3,
    color: Colors.blue,
    child: Center(child: Text('Responsive UI')),
  );
}
```

In this example, the container's width and height adjust based on the screen size.

LayoutBuilder

LayoutBuilder can be used to create layouts that respond to constraints like available width and height. It gives you control over how to render widgets in different screen sizes or orientations.

```
@override
Widget build(BuildContext context) {
  return LayoutBuilder(
    builder: (context, constraints) {
      if (constraints.maxWidth < 600) {
        return Text('Small Screen Layout');
      } else {
        return Text('Large Screen Layout');
      }
    },
  );
}
```

OrientationBuilder

OrientationBuilder allows you to modify the UI based on the device’s orientation (portrait or landscape).

```
@override
Widget build(BuildContext context) {
  return OrientationBuilder(
    builder: (context, orientation) {
      return GridView.count(
        crossAxisCount: orientation == Orientation.portrait ? 2 : 4,
        children: List.generate(10, (index) {
          return Container(
            color: Colors.blue,
            margin: EdgeInsets.all(4),
          );
        }),
      );
    },
  );
}
```

5.7 WORKING WITH MATERIAL DESIGN AND CUPERTINO WIDGETS

Flutter provides two primary design libraries for building beautiful UIs: Material Design and Cupertino.

Material Design Widgets

Material Design is a design language developed by Google and is used in most Android apps. Flutter provides a wide range of Material Design widgets out of the box.

- **Scaffold**: Provides the basic structure for a Material Design layout, including an app bar, body, floating action buttons, and more.
- **Material Components:** Flutter includes Material Design widgets like TextField, ElevatedButton, SnackBar, and Drawer, which help create consistent, functional, and visually appealing apps.
- **Themes:** Flutter’s ThemeData class allows you to customize the app’s visual design, including colors, typography, and shape. You can apply a global theme or modify individual components to adhere to your app’s branding.

Cupertino Widgets

Cupertino widgets are used to create iOS-style interfaces. They follow the iOS design guidelines and are ideal for apps targeting iOS or that want to mimic the native iOS look.

- **CupertinoApp**: The iOS equivalent of MaterialApp, this is used to set up the overall theme and layout for an iOS-styled app.
- **Cupertino Components**: These widgets include CupertinoButton, CupertinoTextField, CupertinoSlider, and many more, giving you access to iOS-native UI components.

5.8 HANDLING USER INPUT WITH FORMS AND VALIDATION

Forms are essential for collecting user data in many apps, such as signing up, logging in, or submitting feedback. Flutter makes it easy to handle forms and validate user input.

Form and TextFormField Widgets

Flutter provides the Form and TextFormField widgets to create forms with multiple input fields. A Form widget serves as a container for form fields, while TextFormField is used for individual text inputs.

```
Form(
  key: _formKey,
  child: Column(
    children: <Widget>[
      TextFormField(
        decoration: InputDecoration(labelText: 'Enter your name'),
        validator: (value) {
          if (value == null || value.isEmpty) {
            return 'Please enter some text';
          }
          return null;
        },
      ),
      TextFormField(
        decoration: InputDecoration(labelText: 'Enter your email'),
        validator: (value) {
          if (value == null || !value.contains('@')) {
            return 'Please enter a valid email';
          }
          return null;
        },
      ),
      ElevatedButton(
        onPressed: () {
```

```
        if (_formKey.currentState!.validate()) {
          // Process data
        }
      },
      child: Text('Submit'),
    ),
  ],
 ),
);
```

Form Validation

Validation in Flutter forms ensures that users input correct information. You can validate fields using the validator function, which checks the value of the input and returns an error message if the validation fails. In the example above, the validator checks if the field is empty or if the email format is incorrect. If validation fails, the TextFormField will display an error message below the field.

Handling Form Submission

You can handle form submission by checking whether all fields pass validation before processing the data:

```
final GlobalKey<FormState> _formKey = GlobalKey<FormState>();

if (_formKey.currentState!.validate()) {
  // All fields are valid, proceed with form submission
}
```

You can also reset the form using:

```
_formKey.currentState!.reset();
```

Input Decorators and UI Customization

Customize the appearance of form fields using InputDecoration. This allows you to add labels, hints, icons, and error messages for better user interaction and a cleaner UI design.

```
TextFormField(
  decoration: InputDecoration(
    labelText: 'Enter your email',
    hintText: 'example@domain.com',
    icon: Icon(Icons.email),
  ),
);
```

5.9 CUSTOMIZING WIDGETS AND THEMES

One of Flutter's most powerful features is the ability to customize every aspect of its UI components. You can easily apply custom themes, colors, and styles to ensure that the app has a consistent look and feel.

Custom Themes

Flutter’s ThemeData allows you to apply a consistent theme across your app, such as colors, font styles, and more.

```
MaterialApp(
  theme: ThemeData(
    primaryColor: Colors.blue,
    accentColor: Colors.orange,
    textTheme: TextTheme(
      headline1: TextStyle(fontSize: 72.0, fontWeight: FontWeight.bold),
      bodyText1: TextStyle(fontSize: 14.0, fontFamily: 'Hind'),
    ),
  ),
  home: MyHomePage(),
);
```

Custom Widgets

Creating your own custom widgets in Flutter is easy. You can create a reusable component by extending the StatelessWidget or StatefulWidget classes.

```
class CustomButton extends StatelessWidget {
  final String label;
  final VoidCallback onPressed;

  CustomButton({required this.label, required this.onPressed});

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: onPressed,
      child: Text(label),
    );
  }
}
```

You can now use this custom button anywhere in your app:

```
CustomButton(
  label: 'Press Me',
  onPressed: () {
    print('Button Pressed!');
  },
)
```

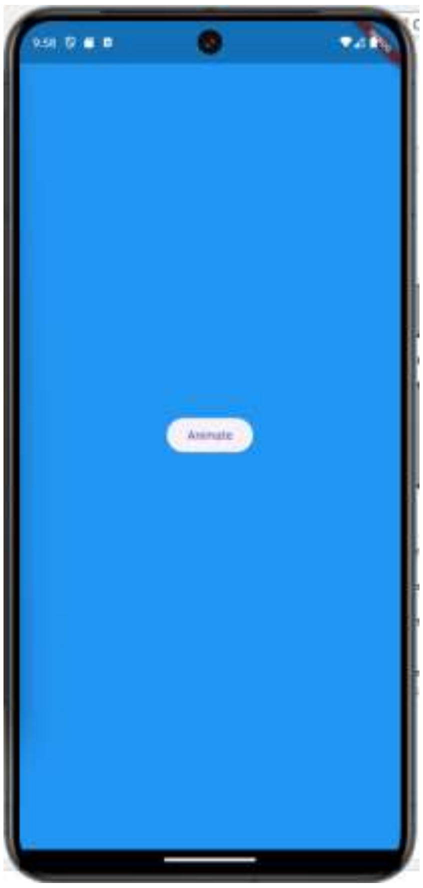
5.10 ANIMATIONS AND TRANSITIONS

Adding animations can enhance the user experience by providing smooth transitions and feedback for user actions. Flutter provides several ways to add animations to your UI.

Implicit Animations

Flutter offers a set of widgets for simple, implicit animations, like AnimatedContainer or AnimatedOpacity. These widgets animate between property changes automatically.

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

void main() {
  runApp(
    const MaterialApp(
      home: AnimatedBox(),
    ),
  );
}

class AnimatedBox extends StatefulWidget {
  const AnimatedBox({super.key});

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

class _AnimatedBoxState extends State<AnimatedBox> {
  double _size = 100;

  void _toggleSize() {
    setState(() {
      _size = _size == 100 ? 200 : 100;
    });
  }

  @override
  Widget build(BuildContext context) {
    return AnimatedContainer(
      width: _size,
      height: _size,
      color: Colors.blue,
      duration: const Duration(seconds: 1),
      curve: Curves.easeInOut,
      child: Center(
          child: ElevatedButton(
              onPressed: _toggleSize, child: const Text('Animate'))),
    );
  }
}
```



Explicit Animations

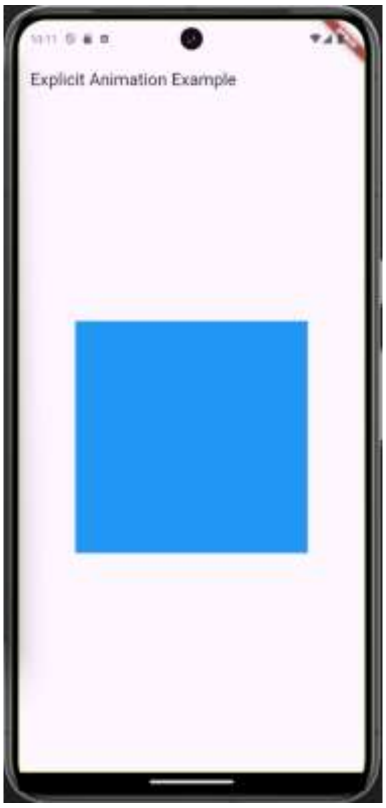
For more control, you can use Flutter’s AnimationController and Tween classes to create explicit animations.

Let’s look at a simple example where a square grows in size over 2 seconds:

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

void main() {
  runApp(
    const MaterialApp(
      home: ExplicitAnimationExample(),
    ),
  );
}

class ExplicitAnimationExample extends StatefulWidget {
  const ExplicitAnimationExample({super.key});

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

class _ExplicitAnimationExampleState extends State<ExplicitAnimationExample>
    with SingleTickerProviderStateMixin {
  late AnimationController _controller;
  late Animation<double> _animation;

  @override
  void initState() {
    super.initState();

    // Step 1: Initialize the AnimationController
    _controller = AnimationController(
      duration: const Duration(seconds: 2),
      vsync: this,
    );

    // Step 2: Define the Tween and connect it with the controller
    _animation = Tween<double>(begin: 0, end: 300).animate(_controller)
      ..addListener(() {
        // Step 3: Rebuild the widget on every tick
        setState(() {});
      });

    // Step 4: Start the animation
    _controller.forward();
  }

  @override
  void dispose() {
    // Dispose of the controller when the widget is removed
    _controller.dispose();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text("Explicit Animation Example")),
      body: Center(
        child: Container(
          width: _animation.value, // The animated size value
          height: _animation.value, // The animated size value
          color: Colors.blue,
        ),
      ),
    );
  }
}
```

Key Points in the Example:

- **AnimationController**: Controls the duration and state of the animation.
- **Tween**: Interpolates values between 0 and 300 for the size of the container.
- **Listener**: Updates the UI on each frame of the animation using setState().
- **Dispose**: The controller is disposed of when the widget is removed from the widget tree to free resources.



Check Your Progress-1

a) MainAxisAlignment aligns widgets along the main axis (vertical for Row and horizontal for Column) True / False

b) CrossAxisAlignment aligns widgets along the cross-axis. (horizontal for Row and vertical for Column). True / False

c) Flutter’s ThemeData allows you to apply a consistent theme across your app, such as colors, font styles, and more. True / False

d) You can use MediaQuery to create adaptive layouts that respond to changes in screen size.

e) Flutter’s ThemeData allows you to apply a consistent theme across your app, such as colors, font styles, and more.

5.11 LET US SUM UP

Flutter offers a vast and flexible toolkit for building rich user interfaces that are responsive, performant, and visually appealing. With its widget-based architecture, you can create highly customizable components, manage state effectively, and add animations that enhance the user experience. Whether you're developing for mobile, web, or desktop, Flutter empowers you to create UIs that look and feel native to each platform.

In this chapter, we've covered the fundamentals of building UIs in Flutter, from basic widgets and layouts to responsive designs and animations. By mastering these tools and techniques, you can craft beautiful, responsive, and engaging applications that will captivate users across devices and platforms.

5.12 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a) False 1-b) False 1-c) True 1-d) True 1-e) True

5.13 FURTHER READING

1. https://docs.flutter.dev/ui/layout
2. https://docs.flutter.dev/ui/adaptive-responsive
3. https://docs.flutter.dev/ui/adaptive-responsive/safearea-mediaquery

5.14 ASSIGNMENT

1. What makes flutter a strong choice for ui development?
2. Explain use of MainAxisAlignment and CrossAxisAlignment properties
3. What is Flexible and Expanded widgets?
4. What is difference between Implicit and Explicit animation?
5. Define: Responsive Layout

Unit-6: State Management in Flutter

6

Unit Structure

6.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Define what state is and explain why state management is critical in Flutter applications.
- Differentiate between local (ephemeral) state and global (app-wide) state.
- Implement basic state management using the setState() method.
- Identify the limitations of setState() in managing complex or global state.
- Describe the core principles and use cases of popular state management libraries like Provider, Riverpod, Bloc, and GetX.
- Compare the features and benefits of each state management approach.
- Build a simple app using Provider to manage global state.
- Understand the role of ChangeNotifier and how to use Consumer to listen to state changes.
- Implement state management using Riverpod and explain the benefits it offers over Provider.
- Use different types of providers in Riverpod to manage app state.
- Implement state management using Bloc and understand the separation of business logic from UI.
- Manage state transitions using events and states in a Bloc pattern.
- Create reactive state variables using GetX and observe automatic UI updates.
- Implement state management, navigation, and dependency injection using GetX.
- Evaluate and choose the appropriate state management solution based on the size and complexity of a project.
- Understand when to use setState(), Provider, Riverpod, Bloc, or GetX in a real-world scenario.

6.2 INTRODUCTION

State management is a fundamental concept in Flutter app development. Understanding how to efficiently manage and update the state of your app is crucial to creating dynamic and responsive user interfaces. This chapter will explore state management in Flutter, from basic techniques using setState() to more advanced approaches using popular state management packages like Provider, Riverpod, Bloc, and GetX. We will also guide you on how to choose the most suitable state management approach for your specific project.

6.3 UNDERSTANDING STATE AND WHY IT MATTERS

In Flutter, **state** refers to any data that affects the appearance or behavior of your app. For example, the state of a counter app is the integer value displayed on the screen. Similarly, the state of a shopping app might include the list of items in a cart.
State in Flutter can be categorized as:

- **Ephemeral State (Local State)**: State that is only relevant to a single widget or a small section of the widget tree, such as the current value of a text field or a toggle switch.
- **App State (Global State)**: State that needs to be shared across multiple parts of the app, such as user authentication status or a theme setting. Global state management is more complex and requires special tools or patterns.

Why does state management matter? When state changes, the user interface must update to reflect those changes. If not handled correctly, state management can lead to unnecessary re-renders, slow performance, and convoluted code.

6.4 LOCAL STATE MANAGEMENT USING setState()

The most basic and commonly used form of state management in Flutter is local state management, which uses the setState() function. This approach is ideal for managing small, transient pieces of state that are confined to a single widget.

Example: Managing Local State with setState()

```
class CounterPage extends StatefulWidget {
  @override
  _CounterPageState createState() => _CounterPageState();
}

class _CounterPageState extends State<CounterPage> {
  int _counter = 0; // Local state

  void _incrementCounter() {
    setState(() {
      _counter++; // Rebuild the UI with the new state
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Counter Example')),
      body: Center(
        child: Text('Counter: $_counter', style: TextStyle(fontSize: 24)),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: _incrementCounter,
        child: Icon(Icons.add),
      ),
    );
  }
}
```

Limitations of setState()

While setState() is simple and effective for managing local state, it has limitations:

- **Reusability**: The state is confined to a specific widget and cannot easily be shared with other parts of the app.
- **Code Clutter**: When managing multiple pieces of state across different parts of the app, using setState() can result in scattered and unmanageable code.
- **Performance**: Overusing setState() can lead to unnecessary widget rebuilds, affecting the performance of large or complex apps.

6.5 OVERVIEW OF POPULAR STATE MANAGEMENT APPROACHES

As apps grow in complexity, you’ll need more robust state management solutions. Flutter offers several powerful libraries and patterns that help manage state more effectively, especially for global state. **Here are four popular state management approaches:**

1. **Provider:** The Provider package is one of the most widely used state management solutions in Flutter. It allows you to share state between widgets in a clean, efficient way and is easy to scale as your app grows. Provider works by

making data available throughout the widget tree and notifying listeners whenever the state changes.

Key Concepts:

- **ChangeNotifier**: A class that provides a simple way to notify listeners about changes in state.
- **Consumer**: A widget that listens for changes in the ChangeNotifier and rebuilds when the state changes.

Example Using Provider:

```
class CounterModel with ChangeNotifier {
  int _counter = 0;

  int get counter => _counter;

  void incrementCounter() {
    _counter++;
    notifyListeners(); // Notify listeners that the state has changed
  }
}

class CounterApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return ChangeNotifierProvider(
      create: (_) => CounterModel(),
      child: MaterialApp(
        home: CounterScreen(),
      ),
    );
  }
}


class CounterScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    final counterModel = Provider.of<CounterModel>(context);

    return Scaffold(
      appBar: AppBar(title: Text('Provider Example')),
      body: Center(
        child: Text('Counter: ${counterModel.counter}', style: TextStyle(fontSize: 24)),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: counterModel.incrementCounter,
        child: Icon(Icons.add),
      ),
    );
  }
}
```

Provider simplifies the sharing of state across multiple widgets and automatically handles notifying listeners when the state changes.

2. **Riverpod:** Riverpod is a modern state management library inspired by Provider but with additional features and better safety. It removes some of the limitations of Provider and offers more flexibility, such as compile-time safety and the ability to handle side effects.

   Key Concepts:
   - **Providers**: Declares a piece of state that can be used and updated across the app.
   - **StateNotifier**: Similar to ChangeNotifier, but more suited for managing state mutations and logic.

   Riverpod works similarly to Provider, but with improvements that make it easier to use in large applications. It's also more type-safe and testable.

3. **Bloc (Business Logic Component):** Bloc is a more structured state management pattern that separates business logic from UI. It uses **Streams** to handle asynchronous data and events, making it a good choice for apps with complex state or requiring advanced asynchronous operations.

   Key Concepts:
   - **Bloc**: Encapsulates business logic and handles state transitions.
   - **Events**: Actions that trigger changes in the Bloc.
   - **States**: The output of the Bloc, representing the current state of the app.

   Bloc enforces a clear separation between business logic and UI, which helps maintainable and testable code.

4. GetX: GetX is an all-in-one package that provides state management, navigation, and dependency injection. It is known for being lightweight and easy to use, making it a popular choice for developers looking for a simple solution that doesn’t require boilerplate code.

Key Concepts:

- **Reactive Programming**: In GetX, state variables are made reactive so that the UI automatically updates when the state changes.
- **Controllers**: Store the state and logic of the app.

Example Using GetX:

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

void main() {
  runApp(MaterialApp(
    home: Scaffold(
      body: Center(
        child: CounterScreen(),
      ),
    ),
  ));
}

class CounterController extends GetxController {
  var counter = 0.obs; // Make counter reactive

  void increment() => counter++;
}

class CounterScreen extends StatelessWidget {
  final CounterController c = Get.put(CounterController());

  CounterScreen({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('GetX Example')),
      body: Center(
        child: Obx(() => Text('Counter: ${c.counter}',
            style: const TextStyle(fontSize: 24))),
      ),
      floatingActionButton: FloatingActionButton(
        onPressed: c.increment,
        child: const Icon(Icons.add),
      ),
    );
  }
}
```

6.6 CHOOSING THE RIGHT STATE MANAGEMENT FOR YOUR PROJECT

Choosing the right state management solution depends on the size and complexity of your project, as well as your team's familiarity with various tools. Here are some guidelines:

For Small Apps or Local State:

- Use setState() for simple state management that’s confined to individual widgets. This is ideal for smaller apps or when only a small portion of the UI needs to change.

For Medium-Sized Apps:

- **Provider** or **Riverpod**: These are well-suited for apps that have moderate complexity and need to share state across multiple widgets or screens. They are easy to set up, scale well, and have extensive community support.

For Large or Complex Apps:

- **Bloc**: If your app has complex business logic, asynchronous data streams, or needs strict separation between logic and UI, Bloc is a good choice. It helps structure your app for long-term scalability and testability.
- **GetX**: Choose GetX for apps where you need a simple but powerful state management solution that integrates well with navigation and dependency injection.

For Apps with Async Operations:

- **Bloc**: Bloc is built around handling asynchronous operations and is ideal for apps with complex API calls, data streams, or event-driven architectures.

For Developer Productivity:

- **GetX**: GetX is highly productive due to its simplicity, minimal boilerplate, and support for various aspects of app development (state, navigation, etc.).

Check your progress-1

a) The most basic and commonly used form of state management in Flutter is local state management, which uses the __________function.

b) ________ state management is more complex and requires special tools or patterns. (local, global)

c) There are _______popular state management approaches

d) Give full form of BLOC

e) ________ is is highly productive due to its simplicity, minimal boilerplate, and support for various aspects of app development

f) State management is a critical part of building Flutter apps. (True/False).

6.7 LET US SUM UP

State management is a critical part of building Flutter apps. Whether you are working on a small app with limited state changes or a large app with complex business logic, understanding the right tools and techniques for managing state will help you build efficient, scalable, and maintainable applications. From basic local state management with setState() to more advanced approaches like Provider, Riverpod, Bloc, and GetX, each solution has its strengths and use cases. By choosing the right state management approach for your project, you can streamline your development process and ensure a smoother user experience.

6.8 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a) setState()

1-b) Global

1-c) four

1-d) Business Logic Component

1-e) GetX

1-f) True

6.9 FURTHER READING

- https://docs.flutter.dev/data-and-backend/state-mgmt/intro
- https://docs.flutter.dev/data-and-backend/state-mgmt/options

6.10 ASSIGNMENT

1. What is state management?
2. Explain local state management in flutter with suitable example
3. What is local state and global state
4. List limitations of setState()
5. Write detailed note on popular state management approaches used in Flutter
6. What are the guidelines for choosing the Right State Management for Your Project?

Unit-7: Working with Flutter Navigation

7

Unit Structure

7.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Explain how Flutter uses the Navigator and routes to manage navigation between screens.
- Define the stack-based navigation system and its role in app navigation.
- Navigate between screens using Navigator.push() to add routes and Navigator.pop() to remove them.
- Identify scenarios where basic navigation is sufficient for app flows.
- Set up named routes in Flutter to streamline navigation across multiple screens.
- Navigate using Navigator.pushNamed() and Navigator.pop() with named routes.
- Pass data between screens using both unnamed and named routes.
- Retrieve and use data passed between screens using ModalRoute.of(context).settings.arguments.
- Explain what deep linking is and why it's important for app navigation.
- Implement deep linking in Flutter using packages like uni_links to handle external links that open specific screens within the app.
- Handle complex navigation flows, such as multi-step workflows, onboarding processes, and conditional navigation.
- Implement dynamic routing based on deep link URIs and manage navigation states accordingly.
- Explain the concept of nested navigation and when it is used.
- Implement nested navigation in apps using widgets like BottomNavigationBar or tab-based navigation, each with independent navigation stacks.
- Describe the limitations of Navigator 1.0 and the benefits of the Navigator 2.0 API.
- Implement navigation using Navigator 2.0 for more complex use cases and web apps, including using the Router widget and declarative navigation.
- Evaluate the complexity of your app’s navigation flow and select the appropriate navigation method (basic navigation, named routes, deep linking, or Navigator 2.0).
- Apply best practices to ensure smooth, efficient, and scalable navigation flows in your Flutter projects.

7.2 INTRODUCTION

Navigation is a fundamental aspect of any mobile application. Whether you're navigating between simple pages, passing data between screens, or handling complex deep links, understanding Flutter's navigation system is essential for creating smooth and intuitive user experiences. In this chapter, we will dive deep into Flutter's navigation system, covering basic navigation using the Navigator and routes, how to use named routes, passing data between screens, handling deep links, managing complex navigation flows, and nested navigation using the advanced Navigator 2.0 API.

7.3 BASIC NAVIGATION USING NAVIGATOR AND ROUTES

In Flutter, navigation between screens is managed by the Navigator class, which works like a stack. When a new screen is pushed onto the stack, it appears on top of the current screen, and when the top screen is popped, the previous screen becomes visible again. The most basic way to navigate between screens in Flutter is by using the Navigator and defining routes.

Basic Navigation with Navigator.push() and Navigator.pop()

To navigate to a new screen, you can use the Navigator.push() method, and to go back to the previous screen, you use Navigator.pop().

Example: Basic Navigation

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Navigation',
      home: HomeScreen(),
    );
  }
}
class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Home Screen')),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            // Navigate to the next screen
            Navigator.push(
              context,
              MaterialPageRoute(builder: (context) => SecondScreen()),
            );
          },
          child: Text('Go to Second Screen'),
        ),
      ),
    );
  }
}

class SecondScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Second Screen')),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            // Go back to the previous screen
            Navigator.pop(context);
          },
          child: Text('Go back'),
        ),
      ),
    );
  }
}
```

In this example, clicking the button on the HomeScreen pushes the SecondScreen onto the stack, and pressing the button on the SecondScreen pops it from the stack, returning to the HomeScreen.

7.4 NAMED ROUTES

For larger apps with multiple screens, managing navigation using unnamed routes can become cumbersome. Flutter provides **named routes** as a way to define and navigate between screens more efficiently.

Setting Up Named Routes

Named routes can be defined in the MaterialApp widget using the routes property, which maps a route name (String) to a widget (screen).

Example: Using Named Routes

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

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

class MyApp extends StatelessWidget {
  const MyApp({super.key});

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Named Routes Example',
      // Define the routes
      routes: {
        '/': (context) => const HomeScreen(),
        '/second': (context) => const SecondScreen(),
      },
    );
  }
}

class HomeScreen extends StatelessWidget {
  const HomeScreen({super.key});

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Home Screen')),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            // Navigate to the second screen using a named route
            Navigator.pushNamed(context, '/second');
          },
          child: const Text('Go to Second Screen'),
        ),
      ),
    );
  }
}

class SecondScreen extends StatelessWidget {
  const SecondScreen({super.key});
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: const Text('Second Screen')),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            // Go back to the previous screen
            Navigator.pop(context);
          },
          child: const Text('Go back'),
        ),
      ),
    );
  }
}
```

In this case, the Navigator.pushNamed() method is used to navigate between named routes. The benefit of using named routes is that they can be easily reused throughout the app and are defined centrally.

7.5 PASSING DATA BETWEEN SCREENS

Often, you'll need to pass data between screens. This can be done by passing arguments when navigating to a screen.

Example: Passing Data Using Named Routes

```
class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Home Screen')),
      body: Center(
        child: ElevatedButton(
          onPressed: () {
            // Pass data to the second screen
            Navigator.pushNamed(
              context,
              '/second',
              arguments: 'Hello from HomeScreen!',
            );
          },
          child: Text('Go to Second Screen with Data'),
        ),
      ),
    );
  }
}

class SecondScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    // Retrieve the data passed as an argument
    final String data = ModalRoute.of(context)!.settings.arguments as String;

    return Scaffold(
      appBar: AppBar(title: Text('Second Screen')),
      body: Center(
        child: Text(data, style: TextStyle(fontSize: 24)),
      ),
    );
  }
}
```

In this example, the arguments parameter is used to pass data from HomeScreen to SecondScreen. The ModalRoute.of(context).settings.arguments is then used to retrieve the data.

7.6 DEEP LINKING AND MANAGING COMPLEX NAVIGATION FLOWS

Deep linking allows your app to be opened directly at a specific screen from external sources, like a web link or another app. This is essential for apps that handle complex navigation flows, such as those that require specific onboarding processes or multi-step workflows.

What is Deep Linking?

Deep linking refers to using a URL to link directly to a specific page in your app. For instance, a user might click on a link in an email that directly opens a product page in an e-commerce app.

Handling Deep Links in Flutter

Flutter provides support for deep linking through packages like uni_links and firebase_dynamic_links. You can handle deep links by parsing incoming URIs and routing the user to the appropriate screen based on the URI.

7.7 NESTED NAVIGATION

In more advanced use cases, apps may need to handle **nested navigation** where a section of the app has its own navigation stack. For example, a tabbed application might have separate navigation flows for each tab.

Nested Navigation

Nested navigation allows different parts of an app to maintain their own navigation history independently. This is common in apps with multiple sections, like a bottom navigation bar or drawer menu.

Example: Nested Navigation with a Bottom Navigation Bar

```
class MyApp extends StatefulWidget {
  @override
  _MyAppState createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  int _currentIndex = 0;

  final List<Widget> _screens = [
    HomeScreen(),
    SettingsScreen(),
  ];

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        body: _screens[_currentIndex],
        bottomNavigationBar: BottomNavigationBar(
          currentIndex: _currentIndex,
          onTap: (index) {
            setState(() {
              _currentIndex = index;
            });
          },
          items: [
            BottomNavigationBarItem(icon: Icon(Icons.home), label: 'Home'),
            BottomNavigationBarItem(icon: Icon(Icons.settings), label: 'Settings'),
          ],
        ),
      ),
    );
  }
}
```

Here, each tab has its own screen, and the navigation between the tabs is independent.

7.8 WORKING WITH THE NAVIGATOR 2.0 API

The original Navigator API (Navigator 1.0) works well for simple cases but has limitations when handling complex navigation flows. Navigator 2.0 introduces more flexibility by decoupling the navigation logic from the user interface. It uses the Router widget and a new declarative API that gives developers more control over navigation, especially when dealing with web applications or integrating with the browser's back button.

Navigator 2.0 is especially powerful for web apps and apps with more complex state-driven navigation.

Check Your Progress-1

a) In Flutter, navigation between screens is managed by the Navigator class, which works like a ________.

   a) Stack b) Queue c) List d) Tree

b) To navigate to a new screen, you can use the Navigator.push() method, and to go back to the previous screen, you use Navigator.pop(). (True/False)

c) For larger apps with multiple screens, managing navigation using unnamed routes can become easy. (True / False)

d) Nested navigation allows different parts of an app to maintain their own navigation history independently.

   a) Inner b) Outer c) Nested d) All of these

e) Deep linking allows your app to be opened directly at a specific screen from external sources, like a web link or another app.(True / False)

7.9 LET US SUM UP

Navigation in Flutter ranges from simple stack-based navigation using Navigator.push() to more complex workflows using deep linking, named routes, and the advanced Navigator 2.0 API. Each method offers different levels of control and flexibility, and choosing the right approach depends on the complexity of your app's navigation needs. Understanding these tools will help you build smooth, intuitive, and efficient navigation flows in your Flutter applications.

7.10 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-a) Stack
1-b) True
1-c) False
1-d) Nested
1-e) True

7.11 FURTHER READING

- https://docs.flutter.dev/ui/navigation
- https://docs.flutter.dev/cookbook/navigation/navigation-basics
- https://docs.flutter.dev/cookbook/navigation/returning-data

7.12 Assignment

1. Explain navigation using navigator and routes
2. Explain use of named touter
3. What is nested navigation?
4. What is deep linking?
5. Explain with code snippet example how can we pass data between screen?

Block-3:

Flutter Application Data Management

Unit-8: Integrating APIs and Working with Data

8

Unit Structure

8.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Comprehend how REST APIs work and the role of the http package in Flutter for making network requests.
- Use the http package to send GET requests to retrieve data from APIs.
- Parse JSON data and use it to update the user interface dynamically.
- Convert raw JSON responses into Dart objects using jsonDecode.
- Display complex JSON data within Flutter widgets like ListView and GridView.
- Implement loading indicators (e.g., CircularProgressIndicator) while waiting for data.
- Manage errors such as network failures or server errors and display appropriate messages to the user.
- Send Data to APIs Using POST, PUT, and DELETE Requests:
- Implement POST requests to send new data to a server (e.g., creating new records).
- Use PUT requests to update existing data and DELETE requests to remove resources on a server.
- Design Flutter apps that interact seamlessly with external data sources.
- Handle real-time updates and user interactions while managing data flow between the app and APIs effectively.

8.2 INTRODUCTION

In modern mobile applications, working with external data sources is critical. Whether you're building a weather app, social media feed, or a to-do list app, you'll likely need to fetch, display, and send data to and from a remote server via APIs. In Flutter, integrating APIs and working with data is seamless thanks to the http package, which allows us to perform RESTful API operations like GET, POST, PUT, and DELETE requests. This chapter will walk you through the process of integrating APIs, handling data, and managing loading states and errors effectively.

8.3 FETCHING DATA FROM REST APIS USING HTTP PACKAGE

APIs serve as a bridge between your app and external data sources or services. To communicate with REST APIs in Flutter, we use the **http** package, which provides simple methods to send HTTP requests.

Adding the http Package

First, you need to add the http package to your project's pubspec.yaml file:

```
dependencies:
  flutter:
    sdk: flutter
  http: ^0.13.3
```

Run flutter pub get to install the package.

Making a GET Request

Fetching data from REST APIs is a common requirement in modern mobile apps. In this section, we'll break down the code example that fetches data from a REST API using Flutter's http package, step by step.

A **GET request** retrieves data from the server. Here's a basic example of how to fetch data from a public API:

Here we go step by step

Step-1: Importing Required Packages

```
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
import 'dart:convert'; // For working with JSON data
```

- flutter/material.dart: Provides essential widgets for creating a Flutter app, like Scaffold, AppBar, and ListView.
- http: This is the **http** package that enables sending and receiving HTTP requests. It allows us to interact with APIs.
- dart:convert: This library is used to decode the JSON response that we receive from the API.

Step-2: Creating the Main Application

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: ApiDataScreen(),
    );
  }
}
```

- The main() function is the entry point of the app. It runs MyApp(), which is the root widget of our application.
- Inside MyApp, a **MaterialApp** widget is used with ApiDataScreen as the home screen. This is the screen where we will fetch and display API data.

Step-3: Creating Stateful Widget for Fetching Data

```
class ApiDataScreen extends StatefulWidget {
  @override
  _ApiDataScreenState createState() => _ApiDataScreenState();
}
```

- ApiDataScreen is a **stateful widget** because we need to manage changes in data (like fetching data from the API).
- The createState() method creates an instance of _ApiDataScreenState which contains the logic for fetching and displaying the data.

Step-4: Defining the State and Variables

```
class _ApiDataScreenState extends State<ApiDataScreen> {
  List<dynamic> _data = [];
```

- _data: A list that will store the data fetched from the API. It is initialized as an empty list.
- We'll later update this list with the data we fetch from the API and display it on the screen.

Step-5: Fetching Data with a GET Request

```
Future<void> fetchData() async {
final response = await
http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts'));
```

- fetchData(): A **future** function that sends a GET request to the API. We use the http.get() method to send the request.
- Uri.parse(): Converts the string URL into a URI format that is needed by the http package.
- The API URL in this example is **https://jsonplaceholder.typicode.com/posts**, which returns a list of placeholder posts.

Step-6: 6. **Handling the API Response**

```
    if (response.statusCode == 200) {
      // Parse the JSON response
      setState(() {
        _data = jsonDecode(response.body);
      });
    } else {
      throw Exception('Failed to load data');
    }
  }
```

- **statusCode == 200**: The status code 200 means that the request was successful.
- **jsonDecode(response.body)**: Converts the response body (which is a JSON string) into a Dart object, which in this case is a list of posts. Each post has fields like id, title, and body.
- **setState()**: Once the data is successfully fetched, we update the _data list with the parsed data inside setState() so that the UI will re-render and display the fetched data.
- **Error handling**: If the API call fails (i.e., the status code is not 200), an exception is thrown to indicate that the data could not be loaded.

Step-7: Calling the API on Widget Creation

```
  @override
  void initState() {
    super.initState();
    fetchData(); // Fetch data when the widget is first created
  }
```

- **initState()**: This lifecycle method is called when the widget is first inserted into the widget tree. We call fetchData() here to ensure that the data is fetched immediately when the screen loads.
- When the data is fetched, the _data list is populated, and the UI will display the data.

Step-8: Building the UI

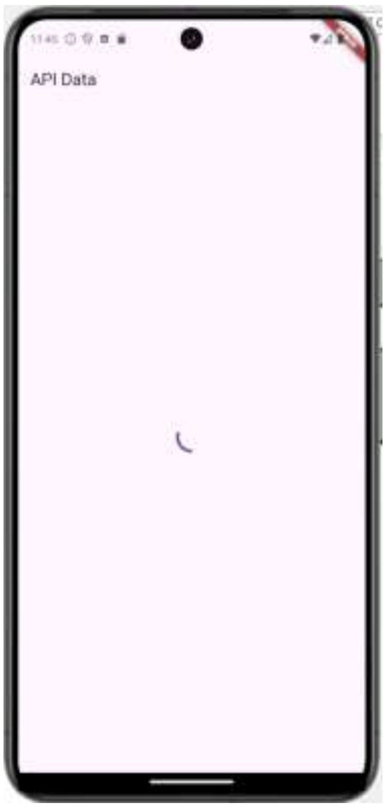
```
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('API Data'),
      ),
      body: _data.isEmpty
          ? Center(child: CircularProgressIndicator()) // Show loading spinner while data
is being fetched
          : ListView.builder(
            itemCount: _data.length,
            itemBuilder: (context, index) {
              return ListTile(
                title: Text(_data[index]['title']),
                subtitle: Text(_data[index]['body']),
              );
            },
          ),
    );
  }
}
```

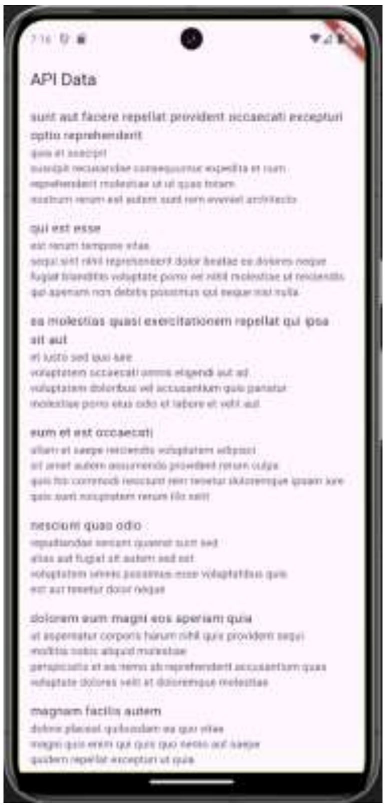
- **Scaffold**: Provides the structure for the screen, including the AppBar and the body of the app.
- **CircularProgressIndicator**: A loading spinner that is displayed while the data is being fetched (i.e., when _data is empty).
- **ListView.builder**: A scrollable list that dynamically generates list items based on the number of data entries (_data.length). For each item in the list, we create a ListTile displaying the title and body of each post.

Breaking Down ListView:

- itemCount: _data.length: Specifies how many list items to build (one for each data entry).
- itemBuilder: Builds each list item with a title and subtitle using data from the _data list.

This example demonstrates how to perform basic API calls and display data in a Flutter app, which is essential for creating dynamic, data-driven applications.




API Data

sunt aut facere repellat provident occaecati excepturi optio reprehenderit
quia et suscipit
suscipit recusandae consequuntur expedita et cum
reprehenderit molestiae ut ut quas totam
nostrum rerum est autem sunt rem eveniet architecto

qui est esse
est rerum tempore vitae
sequi sint nihil reprehenderit dolor beatae ea dolores neque
fugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis
qui aperiam non debitis possimus qui neque nisi nulla

ea molestias quasi exercitationem repellat qui ipsa sit aut
et iusto sed quo iure
voluptatem occaecati omnis eligendi aut ad
voluptatem doloribus vel accusantium quis pariatur
molestiae porro eius odio et labore et velit aut

eum et est occaecati
ullam et saepe reiciendis voluptatem adipisci
sit amet autem assumenda provident rerum culpa
quis hic commodi nesciunt rem tenetur doloremque ipsam iure
quis sunt voluptatem rerum illo velit

nesciunt quas odio
repudiandae veniam quaerat sunt sed
alias aut fugiat sit autem sed est
voluptatem omnis possimus esse voluptatibus quis
est aut tenetur dolor neque

dolorem eum magni eos aperiam quia
ut aspernatur corporis harum nihil quis provident sequi
mollitia nobis aliquid molestiae
perspiciatis et ea nemo ab reprehenderit accusantium quas
voluptate dolores velit et doloremque molestiae

magnam facilis autem
dolore placeat quibusdam ea quo vitae
magni quis enim qui quis quo nemo aut saepe
quidem repellat excepturi ut quia

8.4 PARSING JSON DATA AND DISPLAYING IT IN THE UI

APIs typically return data in **JSON (JavaScript Object Notation)** format. JSON is a lightweight format for storing and transporting data, which is easy to parse into Flutter objects.

In the above example, the data returned from the API is in JSON format and is decoded using jsonDecode(response.body).

Example of JSON Data:

Here's an example of what JSON data might look like when returned from an API:

json

Copy code

```
[
  {
    "userId": 1,
    "id": 1,
    "title": "First Post Title",
    "body": "This is the content of the first post."
  },
  {
    "userId": 1,
    "id": 2,
    "title": "Second Post Title",
    "body": "This is the content of the second post."
  }
]
```

In the example above, each item in the list has a title and a body, which we display using ListTile in our app.

8.5 HANDLING LOADING STATES AND ERROR CASES

When fetching data from an API, we need to handle potential delays in response (loading state) as well as errors (e.g., server issues or network problems).

Loading State

The most common way to indicate loading is by displaying a **loading spinner** (e.g., CircularProgressIndicator). In the example above, we show the spinner if the _data list is empty:

```
body: _data.isEmpty
    ? Center(child: CircularProgressIndicator())
    : ListView.builder(...);
```

Error Handling

It’s also important to handle scenarios where the API request fails, such as a timeout or a server returning a 500 error. You can show an error message when this happens:

```
Future<void> fetchData() async {
  try {
    final response = await
http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts'));

    if (response.statusCode == 200) {
      setState(() {
        _data = jsonDecode(response.body);
      });
    } else {
      throw Exception('Failed to load data');
    }
  } catch (e) {
    setState(() {
      _data = [];
    });
    // Show an error message or dialog
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(content: Text('Failed to load data: $e')),
    );
  }
}
```

Here, we use a try-catch block to catch any errors that may occur during the API call and display a message using SnackBar.

8.6 SENDING DATA TO APIS (POST, PUT, DELETE REQUESTS)

APIs not only allow you to fetch data but also to **send data** to servers. You may need to create new entries (POST), update existing data (PUT), or delete entries (DELETE). Here's how you can perform these requests in Flutter.

POST Request: Creating New Data

A **POST request** is used to send new data to the server. Here's how you can send data to an API using the http.post method:

```
Future<void> createPost(String title, String body) async {
  final response = await http.post(
    Uri.parse('https://jsonplaceholder.typicode.com/posts'),
    headers: {"Content-Type": "application/json"},
    body: jsonEncode({
      "title": title,
      "body": body,
      "userId": 1,
    }),
  );

  if (response.statusCode == 201) {
    print('Post created successfully: ${response.body}');
  } else {
    throw Exception('Failed to create post');
  }
}
```

In this example, we're sending a new post with a title, body, and userId to the server. The API will process this data and, upon success, return a 201 status code indicating that the resource has been created.

PUT Request: Updating Existing Data

To update an existing record, we use the **PUT request**. This is similar to a POST request but is used for modifying existing resources:

```
Future<void> updatePost(int id, String title, String body) async {
  final response = await http.put(
    Uri.parse('https://jsonplaceholder.typicode.com/posts/$id'),
    headers: {"Content-Type": "application/json"},
    body: jsonEncode({
      "title": title,
      "body": body,
      "userId": 1,
    }),
  );

  if (response.statusCode == 200) {
    print('Post updated successfully: ${response.body}');
  } else {
    throw Exception('Failed to update post');
  }
}
```

Here, we specify the id of the post we want to update and send the updated title and body to the API.

DELETE Request: Deleting Data

A **DELETE request** is used to remove a resource from the server. Here's an example:

```
Future<void> deletePost(int id) async {
  final response = await http.delete(
    Uri.parse('https://jsonplaceholder.typicode.com/posts/$id'),
  );

  if (response.statusCode == 200) {
    print('Post deleted successfully');
  } else {
    throw Exception('Failed to delete post');
  }
}
```

Check Your Progress

1. Which package is commonly used in Flutter for making HTTP requests to REST APIs?

   a) json b) http c) flutter_http d) api_flutter

2. What method is used to decode a JSON response in Flutter?

   a) jsonDecode() b) jsonEncode() c) decodeJson() d) toJson()

3. In the following code, which HTTP method is being used to send data to an API?

```
final response = await http.post(
    Uri.parse('https://example.com/api'),
    headers: {"Content-Type": "application/json"},
    body: jsonEncode({"name": "John", "age": 30}),
);
```

   a) GET b) POST c) PUT d) DELETE

4. What widget can you use in Flutter to display a loading spinner while data is being fetched?

   a) ListView b) Scaffold c) CircularProgressIndicator d) LinearProgressIndicator

5. If an API returns a status code of 200, what does it signify?

   a) The request failed due to a client error

   b) The request was successful

   c) The server is down

   d) The request is unauthorized

8.7 LET US SUM UP

In this chapter, we explored how to integrate REST APIs into your Flutter applications using the http package. We covered how to:

1. Fetch and display data using **GET requests**.
2. Parse **JSON data** and render it in the UI.
3. Handle **loading states** and manage errors effectively.
4. Send data to APIs using **POST, PUT, and DELETE requests**.

By mastering these techniques, you'll be able to create feature-rich applications that communicate with external servers and provide dynamic, real-time content to users.

8.8 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1. b) http
2. a) jsonDecode()
3. b) POST
4. c) CircularProgressIndicator
5. b) The request was successful

8.9 FURTHER READING

1. https://docs.flutter.dev/data-and-backend/networking
2. https://docs.flutter.dev/cookbook/networking/fetch-data

8.10 ASSIGNMENT

1. Explain the role of the http package in Flutter. Why is it essential for working with APIs?
2. Describe the process of converting a JSON response from an API into Dart objects. Why is jsonDecode() necessary?
3. What is the difference between GET and POST requests in the context of REST APIs? Provide examples of when each would be used in a Flutter app.
4. How would you handle loading and error states when fetching data from an API in Flutter? Why is it important to manage these states effectively?
5. Describe a real-world scenario where you would need to use a PUT or DELETE request in a Flutter app. How would you implement this functionality?

8.11 ACTIVITIES

1. **Fetch Data from an API and Display It in a Flutter App**
   Create a Flutter application that fetches data from a public REST API (e.g., a list of posts or users from the JSONPlaceholder API) using the http package. Display the fetched data in a ListView widget and ensure that the app shows a loading spinner while data is being fetched.
2. **Handling Loading and Error States**

Modify your existing Flutter app to handle loading and error states more effectively. Add a loading indicator when data is being fetched, and display an error message if the API request fails (e.g., due to network issues or an invalid URL). Document your approach to managing both loading and error states.

3. **Parsing and Displaying Nested JSON Data**

   Find or create an API that returns a more complex, nested JSON structure (e.g., a list of users where each user has nested attributes such as address, company, etc.). Parse this nested data in your Flutter app and display key pieces of information in the UI. Ensure that you properly handle nested objects and lists when parsing the JSON.

4. **Sending Data to a REST API**

   Build a Flutter form that allows users to input data (e.g., name, email, message). When the form is submitted, send this data to an API endpoint using a POST request. If a valid API is unavailable, simulate this process using a dummy REST API like Reqres for testing.

5. **Update and Delete Data Using REST API (PUT and DELETE)**

   Create a Flutter app that can update and delete data from a REST API using PUT and DELETE requests. Use a public API or mock API to test your implementation. Demonstrate how to update existing records (e.g., modify a user's information) and delete records (e.g., remove a post or user) from the server.

Unit-9: Storing Data Locally 9

Unit Structure

9.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Identify and differentiate between various local storage solutions such as Shared Preferences, SQLite, and Hive, and understand their use cases and limitations.
- Set up and manage a SQLite database in a Flutter app using the sqflite package, perform CRUD (Create, Read, Update, Delete) operations, and handle structured data efficiently.
- Integrate Hive into a Flutter application for fast, NoSQL data storage and implement data models for efficient local storage of unstructured or large datasets.
- Utilize the Shared Preferences package to store simple key-value pairs, manage user preferences, and retain app state or settings across sessions.
- Evaluate the advantages and disadvantages of Shared Preferences, SQLite, and Hive in various application scenarios and choose the appropriate storage solution based on performance, data structure, and persistence needs.
- Understand the importance of local storage for offline applications, and how to manage and retrieve data even when the app is not connected to the internet.

9.2 INTRODUCTION

In mobile app development, managing data is crucial, especially when dealing with offline scenarios or when you need to persist user data between app sessions. Flutter provides several options for **local storage**, enabling you to store and retrieve data efficiently without always relying on a remote server. In this chapter, we will explore the key topics such as Introduction to Local Storage Options: Shared Preferences, SQLite, and Hive, Implementing Persistent Storage with SQLite, Using Hive for Lightweight, High-Performance Local Storage and Managing App Settings and Preferences.

9.3 INTRODUCTION TO LOCAL STORAGE OPTIONS

Flutter offers multiple ways to store data locally, each suited for different use cases. Let’s explore three common local storage options:

a) Shared Preferences

Shared Preferences is a simple key-value storage system. It is used for storing small amounts of primitive data types, such as strings, booleans, and integers. The data is stored in a file on the device and can be retrieved even after the app is closed and reopened.

- **Use case**: Store user preferences, theme settings, or app state (e.g., whether the user has logged in).
- **Advantages**: Easy to use, minimal setup, good for lightweight data.
- **Disadvantages**: Not suitable for storing large or complex data.

b) SQLite

SQLite is a lightweight, embedded relational database. It provides powerful querying capabilities with SQL syntax and is ideal for structured data. It can handle more complex data structures than Shared Preferences.

- **Use case**: Store structured data like user profiles, products, or application content.
- **Advantages**: Supports complex queries, relational data, and multiple tables.
- **Disadvantages**: Requires more setup and management compared to Shared Preferences.

c) Hive

Hive is a modern, NoSQL database for Flutter. It is designed for high-performance storage and works well for apps that require fast and efficient data access. Hive stores data in a binary format, making it significantly faster than SQLite for certain types of data.

- **Use case**: Store large amounts of unstructured data like user settings, preferences, or cached content.
- **Advantages**: Very fast, minimal setup, no need for a relational schema.
- **Disadvantages**: Limited query capabilities compared to SQLite.

9.4 IMPLEMENTING PERSISTENT STORAGE WITH SQLITE

SQLite is a powerful and widely used database in mobile applications. In Flutter, you can use the sqflite package to work with SQLite databases. Below is an example of how to implement SQLite for local storage.

a) Adding the sqflite package

First, include the sqflite package and path_provider in your pubspec.yaml file:

```
dependencies:
  sqflite: ^2.0.0
  path_provider: ^2.0.0
```

b) Setting up SQLite Database

```
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';

class DatabaseHelper {
  static final DatabaseHelper _instance = DatabaseHelper._internal();
  factory DatabaseHelper() => _instance;

  static Database? _database;

  DatabaseHelper._internal();

  Future<Database> get database async {
    if (_database != null) return _database!;
    _database = await _initDatabase();
    return _database!;
  }

  Future<Database> _initDatabase() async {
    String path = join(await getDatabasesPath(), 'app_data.db');
    return await openDatabase(
      path,
      version: 1,
      onCreate: (db, version) async {
        await db.execute('''
          CREATE TABLE users(
            id INTEGER PRIMARY KEY AUTOINCREMENT,
            name TEXT,
            age INTEGER
          )
        ''');
      },
    );
  }
}
```

c) Inserting and Retrieving Data

```
Future<void> insertUser(Map<String, dynamic> user) async {
  final db = await DatabaseHelper().database;
  await db.insert('users', user);
}

Future<List<Map<String, dynamic>>> getUsers() async {
  final db = await DatabaseHelper().database;
  return await db.query('users');
}
```

d) Displaying Data in the UI

You can now display the fetched data in a Flutter widget like ListView:

```
FutureBuilder<List<Map<String, dynamic>>>(
  future: getUsers(),
  builder: (context, snapshot) {
    if (!snapshot.hasData) return CircularProgressIndicator();
    return ListView.builder(
      itemCount: snapshot.data?.length ?? 0,
      itemBuilder: (context, index) {
        return ListTile(
          title: Text(snapshot.data![index]['name']),
          subtitle: Text('Age: ${snapshot.data![index]['age']}'),
        );
      },
    );
  },
)
```

9.5 USING HIVE FOR LIGHTWEIGHT, HIGH-PERFORMANCE LOCAL STORAGE

Hive is an efficient NoSQL database designed for high performance. It is ideal for Flutter apps that need to store large amounts of data quickly.

a) Adding the Hive Package

Add Hive and its Flutter adapter to your pubspec.yaml:

```
dependencies:
  hive: ^2.0.0
  hive_flutter: ^1.1.0
```

b) Initializing Hive

Before using Hive, initialize it in your main() function:

```
import 'package:hive/hive.dart';
import 'package:hive_flutter/hive_flutter.dart';

void main() async {
  await Hive.initFlutter();
  runApp(MyApp());
}
```

c) Storing and Retrieving Data

To store data, first open a box (a container for data storage), then use put() to insert data and get() to retrieve it.

```
var box = await Hive.openBox('settings');

// Storing data
box.put('username', 'JohnDoe');

// Retrieving data
String username = box.get('username');
```

d) Working with Hive Data Models

Hive allows you to store custom objects using Hive models:

```
@HiveType(typeId: 0)
class User extends HiveObject {
  @HiveField(0)
  String name;
  @HiveField(1)
  int age;

  User(this.name, this.age);
}
```

After creating the model, register the adapter:

```
Hive.registerAdapter(UserAdapter());
```

9.6 MANAGING APP SETTINGS AND PREFERENCES WITH SHARED PREFERENCES

For managing user settings, theme modes, or preferences, **Shared Preferences** is an ideal choice. It stores data in key-value pairs and is persistent across app launches.

a) Adding Shared Preferences Package

Add the package in pubspec.yaml:

```yaml
dependencies:
  shared_preferences: ^2.0.0
```

b) Storing and Retrieving Preferences

You can store and retrieve user preferences like theme settings or login status using SharedPreferences.

```dart
import 'package:shared_preferences/shared_preferences.dart';

Future<void> saveLoginStatus(bool isLoggedIn) async {
  final prefs = await SharedPreferences.getInstance();
  prefs.setBool('isLoggedIn', isLoggedIn);
}

Future<bool> getLoginStatus() async {
  final prefs = await SharedPreferences.getInstance();
  return prefs.getBool('isLoggedIn') ?? false;
}
```

Check Your Progress

- Which of the following is the best local storage solution in Flutter for storing simple key-value pairs?
  a) SQLite b) Hive c) Shared Preferences d) Firebase

- Which package is commonly used in Flutter to implement an SQLite database for storing structured data?
  a) sqflite b) hive c) shared_preferences d) path_provider

- What is a primary advantage of using Hive for local storage in Flutter?
  a) It is a relational database with complex querying.
  b) It supports high-performance, NoSQL data storage.

c) It automatically syncs data with a remote server.

d) It is designed to store only key-value pairs.

- What method would you use to save a boolean value, such as a login status, in Shared Preferences?

  a) prefs.setInt() b) prefs.setString() c) prefs.setBool() d) prefs.setDouble()

- Which of the following is true about SQLite in Flutter?

  a) SQLite is a NoSQL database that stores data in a binary format.

  b) SQLite supports complex queries and relational data structures.

  c) SQLite is used only for storing simple key-value pairs.

  d) SQLite automatically syncs data with a cloud server.

9.7 LET US SUM UP

Flutter offers a wide range of local storage solutions tailored for different types of data. **Shared Preferences** is excellent for storing simple key-value pairs, **SQLite** is suitable for relational databases and complex queries, while **Hive** offers high performance for unstructured data. Understanding these storage methods will help you build robust applications that handle offline capabilities, user preferences, and persistent data efficiently.

By mastering these tools, you can create feature-rich Flutter applications that effectively manage and store data locally.

9.8 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-c) Shared Preferences

2-a) sqflite

3-b) It supports high-performance, NoSQL data storage.

4-c) prefs.setBool()

5-b) SQLite supports complex queries and relational data structures.

9.9 FURTHER READING

- https://pub.dev/packages/hive_flutter/versions

- https://pub.dev/packages/shared_preferences
- https://docs.flutter.dev/cookbook/persistence/key-value

9.10 ASSIGNMENT

- What are the main differences between Shared Preferences, SQLite, and Hive? In what scenarios would you use each of them?
- Explain how you would set up a SQLite database in Flutter using the sqflite package. What steps are involved in creating a table and inserting data into it?
- How does Hive differ from SQLite in terms of performance and data storage structure? When would Hive be a better choice than SQLite?
- Describe how Shared Preferences work. What types of data can be stored using Shared Preferences and how can you retrieve them in Flutter?
- In a Flutter app, how would you handle updating app settings (like theme preference) and ensure the settings are persistent across app launches using Shared Preferences?
- Discuss the challenges of managing data locally in offline scenarios and how Flutter's local storage solutions can help overcome these challenges.
- How would you handle data retrieval and display from a SQLite database in a Flutter application? Provide a brief explanation of how to manage reading and writing operations asynchronously.
- Compare the advantages and disadvantages of using Hive over other local storage options like SQLite or Shared Preferences.

9.11 ACTIVITIES

- **Implement User Preferences with Shared Preferences**

  Build a Flutter app that allows users to toggle between light and dark themes. Use the **Shared Preferences** package to store the user's theme preference so that it persists across app sessions. Ensure that the app loads the saved theme preference when reopened.
- **Create a To-Do List App with SQLite**

  Develop a simple to-do list app using **SQLite** as the local database. Implement CRUD (Create, Read, Update, Delete) functionality to allow users to add, view, edit, and delete tasks. Ensure that tasks are saved locally so they remain even after the app is closed and reopened.

- **Storing and Retrieving User Data with Hive**
  Create a Flutter app that allows users to input their profile information (e.g., name, age, and bio). Use **Hive** to store the user data locally and display it in the app. Ensure the data is retrieved and displayed whenever the app is launched again.
- **Managing Complex Data with SQLite**
  Build an app that stores user data with relationships between tables (e.g., a list of books and authors, where each book has an associated author). Use **SQLite** to create two related tables and write SQL queries to retrieve books along with their respective authors. Display the data in a structured format using ListView or DataTable in Flutter.
- **Data Encryption with Hive**
  Create a secure note-taking app using **Hive**. Encrypt the data stored in Hive to protect sensitive information. Implement features to add, edit, and delete notes. Ensure that data encryption is applied when storing and retrieving data from the local storage.
- **Hybrid Storage Solution**
  Develop a Flutter app that uses both **Shared Preferences** and **SQLite**. Store simple key-value pairs such as login status and theme preference in Shared Preferences, while using SQLite to manage more complex data like user profiles and transaction histories. Explain why you chose each storage method for different data types.
- **Persistent Data with Local Notifications**
  Build a Flutter app that stores reminders locally using **SQLite** or **Hive**. Integrate **local notifications** to remind users of their tasks based on the stored data. Ensure that the app retrieves and updates the notification schedule whenever the app is launched.

Unit-10: Working with Firebase in Flutter

10

Unit Structure

10.1 LEARNING OBJECTIVES

By the end of this unit, learner will be able to:

- Explain the core features of Firebase and how it can be used in mobile app development.
- Identify the benefits of using Firebase services like authentication, Firestore, and Cloud Messaging in Flutter applications.
- Set up Firebase Authentication to enable secure user login and sign-out functionality.
- Implement various sign-in methods, including email/password authentication in a Flutter app.
- Set up Firebase Firestore in a Flutter project and perform basic CRUD (Create, Read, Update, Delete) operations.
- Fetch real-time data from Firestore and understand how Firestore syncs data across devices.
- Set up Firebase Cloud Messaging to send push notifications in a Flutter app.
- Configure foreground and background message handlers and manage user permissions for notifications.
- Combine Firebase Authentication, Firestore, and FCM to create a seamless user experience in your mobile applications.
- Develop scalable and secure mobile apps by leveraging Firebase's cloud infrastructure.

10.2 INTRODUCTION

In today's mobile application development landscape, Firebase has emerged as one of the most powerful and versatile platforms for building scalable and feature-rich apps. Flutter, a UI toolkit from Google for building natively compiled applications for mobile, web, and desktop from a single codebase, pairs perfectly with Firebase. This chapter explores how Firebase can enhance your Flutter apps, focusing on integrating Firebase Authentication, Firestore (Firebase's cloud-based database), and Firebase Cloud Messaging for push notifications.

10.3 OVERVIEW OF FIREBASE AND ITS BENEFITS FOR MOBILE APPS

Firebase is a comprehensive platform developed by Google to help developers build high-quality apps quickly and easily. Firebase provides a suite of tools for authentication, real-time database, cloud storage, analytics, crash reporting, machine learning, and more. It reduces the need for back-end development, allowing developers to focus on building user-facing features.

Key Benefits of Firebase for Mobile Apps:

- **Real-time Database**: Firebase Realtime Database and Firestore allow developers to store and sync data across all clients in real-time.
- **Scalability**: Firebase services are highly scalable, enabling apps to handle a growing number of users and data without complicated backend scaling.
- **Authentication**: Firebase Authentication supports various methods, including email/password, Google, Facebook, and anonymous sign-in, making it easy to manage user accounts.
- **Analytics and Crash Reporting**: Firebase provides detailed app usage and crash analytics to help monitor app performance and user behavior.
- **Cloud Messaging**: Firebase Cloud Messaging (FCM) allows you to send push notifications to engage with users and enhance user experience.

By integrating Firebase with Flutter, developers can quickly build feature-rich, cross-platform applications with minimal back-end setup.

10.4 INTEGRATING FIREBASE AUTHENTICATION FOR USER LOGIN

User authentication is a core part of most mobile applications, and Firebase makes it easy to implement secure, reliable authentication in Flutter apps. Firebase Authentication supports various sign-in methods, including email and password, social logins (Google, Facebook, etc.), and anonymous sign-in.

Steps to Implement Firebase Authentication in Flutter:

1. **Set up Firebase in the Flutter Project**:
   - Create a Firebase project on the Firebase Console.
   - Add Android or iOS app credentials to Firebase by following the guided steps in the console.
   - Download and configure the google-services.json file (for Android) or GoogleService-Info.plist (for iOS).

2. **Add Firebase Authentication to Flutter**: In your Flutter project's pubspec.yaml, add the Firebase dependencies for authentication:

   ```
   dependencies:
     firebase_core: latest_version
     firebase_auth: latest_version
   ```

   Run flutter pub get to install these dependencies.

3. **Initialize Firebase in Your Flutter App**: In the main.dart file, initialize Firebase at the start of your app:

   ```
   void main() async {
     WidgetsFlutterBinding.ensureInitialized();
     await Firebase.initializeApp();
     runApp(MyApp());
   }
   ```

4. **Implement Sign-in Functionality**: Add a function for user sign-in with email and password:

   ```
   Future<User?> signInWithEmailAndPassword(String email, String password)
   async {
     try {
       UserCredential userCredential = await FirebaseAuth.instance
           .signInWithEmailAndPassword(email: email, password: password);
       return userCredential.user;
     } catch (e) {
       print(e);
       return null;
     }
   }
   ```

5. **Sign-Out and User Management**: Firebase also provides methods to sign out users and manage authenticated sessions:

```
FirebaseAuth.instance.signOut();
```

With this setup, you can now authenticate users with Firebase Authentication, allowing them to securely log in and out of your Flutter app.

10.5 WORKING WITH FIREBASE FIRESTORE FOR CLOUD DATABASE

Firebase Firestore is a NoSQL, cloud-hosted database that stores and syncs data between users in real time. Firestore offers more flexibility, scalability, and query capabilities than Firebase’s older Realtime Database.

Steps to Use Firestore in Flutter:

1. **Add Firestore to Your Flutter Project**: In your pubspec.yaml, add the Firestore dependency:

```
dependencies:
  cloud_firestore: latest_version
```

2. **Creating a Firestore Instance**: Firestore needs to be initialized like Firebase Auth:

```
FirebaseFirestore firestore = FirebaseFirestore.instance;
```

3. **Adding Data to Firestore**: You can store structured data in Firestore by creating a collection and adding documents to it:

```
CollectionReference users = FirebaseFirestore.instance.collection('users');

Future<void> addUser() {
  return users
      .add({
        'name': 'John Doe',
        'email': 'john@example.com',
        'age': 30
      })
      .then((value) => print("User Added"))
      .catchError((error) => print("Failed to add user: $error"));
}
```

4. **Fetching Data from Firestore**: To fetch data from Firestore, you can listen to real-time changes using streams:

```
StreamBuilder(
  stream: FirebaseFirestore.instance.collection('users').snapshots(),
  builder: (context, AsyncSnapshot<QuerySnapshot> snapshot) {
    if (!snapshot.hasData) return Text('Loading...');
    return ListView(
      children: snapshot.data!.docs.map((document) {
        return ListTile(
          title: Text(document['name']),
          subtitle: Text(document['email']),
        );
      }).toList(),
    );
  },
);
```

Firestore makes data synchronization between devices and platforms seamless, with the added advantage of real-time updates.

4. Implementing Push Notifications with Firebase Cloud Messaging (FCM)

Firebase Cloud Messaging (FCM) allows developers to send notifications and messages to users, even when the app is not running. It can be used to notify users of new content, promotions, or updates.

Steps to Implement FCM in Flutter:

1. **Add FCM to Your Flutter Project**: In your pubspec.yaml, add the FCM dependency:

   ```
   dependencies:
     firebase_messaging: latest_version
   ```

2. **Configure FCM in Firebase Console**: Set up Firebase Cloud Messaging by enabling it in your Firebase project.
3. **Initialize Firebase Messaging in Your App**: Initialize Firebase Messaging in the main.dart file:

   ```
   FirebaseMessaging messaging = FirebaseMessaging.instance;
   ```

4. **Request Permission for Notifications**: Ask the user for notification permissions:

```
await FirebaseMessaging.instance.requestPermission(
  alert: true,
  badge: true,
  sound: true,
);
```

5. **Handling Messages**: Set up background and foreground message handlers:

```
FirebaseMessaging.onMessage.listen((RemoteMessage message) {
  print('Got a message whilst in the foreground!');
  print('Message data: ${message.data}');
  if (message.notification != null) {
    print('Message also contained a notification: ${message.notification}');
  }
});
```

6. **Sending Notifications**: Notifications can be sent through the Firebase Console or programmatically using Firebase's Admin SDK.

   With Firebase Cloud Messaging, you can enhance user engagement by delivering timely push notifications to your app's users.

Check Your Progress-1

1. What is Firebase primarily used for in mobile applications?
   A) Creating user interfaces
   B) Managing app analytics and hosting images
   C) Providing a cloud-based backend infrastructure with real-time database, authentication, and notifications
   D) Writing custom code for app deployment
2. Which Firebase service is used to manage user sign-in and authentication in Flutter apps?
   A) Firebase Firestore
   B) Firebase Cloud Messaging
   C) Firebase Analytics
   D) Firebase Authentication

3. How can data from Firebase Firestore be fetched in real-time in a Flutter app?

   A) By using FutureBuilder

   B) By using StreamBuilder

   C) By using ListView

   D) By using TextEditingController

4. What must be added to your Flutter project to enable push notifications using Firebase Cloud Messaging (FCM)?

   A) cloud_firestore dependency

   B) firebase_messaging dependency

   C) firebase_analytics dependency

   D) firebase_crashlytics dependency

5. Firebase Authentication allows users to sign in using both email/password and third-party providers like Google and Facebook. (TRUE/FALSE)

6. Firestore in Firebase is a SQL-based database optimized for relational data structures.

10.6 LET US SUM UP

Integrating Firebase into your Flutter app provides a powerful and scalable solution for common app features such as user authentication, real-time database management, and push notifications. By leveraging Firebase's tools, you can significantly reduce development time while maintaining a robust and efficient mobile app. Whether it's authenticating users, storing and syncing data, or delivering messages and notifications, Firebase simplifies backend management, letting you focus more on building a polished and engaging user experience.

10.7 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-C, 2-D, 3-B, 4-B, 5-True, 6-False

10.8 FURTHER READING

- https://docs.flutter.dev/data-and-backend/firebase
- https://firebase.google.com/docs/flutter/setup?platform=android

10.9 ASSIGNMENT

- What advantages does Firebase provide when developing mobile applications compared to building your own backend infrastructure?
- Explain the steps required to integrate Firebase Authentication into a Flutter app. What are the common sign-in methods supported by Firebase?
- How would you structure and store user data in Firebase Firestore? Can you describe the differences between collections and documents in Firestore?
- How does real-time data synchronization work in Firebase Firestore, and how does it benefit mobile app users?
- Describe how Firebase Cloud Messaging (FCM) can be used to send push notifications in a Flutter app. What steps are required to configure FCM for both Android and iOS platforms?
- What considerations should be taken when managing user permissions for push notifications in a mobile app using FCM?
- In what scenarios would you choose to use Firebase Firestore over Firebase Realtime Database? What are the key differences between the two?
- What security measures are provided by Firebase to ensure that user authentication and database interactions are secure?
- How would you handle background notifications in a Flutter app with Firebase Cloud Messaging? Can you explain the difference between foreground and background message handling?
- Why is Firebase a good choice for cross-platform mobile development, especially when using Flutter? What are some potential limitations or challenges you might face?

10.10 ACTIVITIES

1. Create a new Firebase project in the Firebase Console., Add both Android and iOS apps to the project. Download and configure the google-services.json and GoogleService-Info.plist files.
2. Set up a login screen that includes fields for email and password. Implement user registration functionality with Firebase Authentication. Create a logout button and test the sign-out process.

3. Design a simple Firestore database structure for a To-Do list (tasks collection). Implement CRUD (Create, Read, Update, Delete) operations for tasks. Use Firestore snapshots to display real-time updates on the tasks list.
4. Configure Firebase Cloud Messaging in the Firebase Console. Request permission for notifications in your Flutter app. Implement a feature to send notifications for task reminders.
5. Create a visually appealing UI for the login, registration, and tasks screens. Implement design elements such as buttons, text fields, and lists. Add error handling and user feedback messages.
6. Test all functionalities of the application thoroughly (authentication, Firestore operations, push notifications). Debug any issues that arise during testing and document the solutions.

Block-4:

Advanced Flutter

Unit-11: Building and Deploying Multi-Platform Applications

11

Unit Structure

11.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- explain how Flutter supports the development of applications across multiple platforms (iOS, Android, Web, and Desktop). They will understand the advantages of using a single codebase for diverse platforms and how Flutter's architecture enables consistent performance and UI across devices.
- utilize platform channels to write platform-specific code in Flutter. They will be able to implement third-party plugins and create custom platform-specific functionality that enhances their applications.
- debug and test their Flutter applications effectively across various platforms. They will learn about different testing strategies and tools available in Flutter to ensure the quality and reliability of their applications.
- prepare and deploy their Flutter applications to the Google Play Store, Apple App Store, and the web. They will learn to navigate the guidelines and requirements for each platform to ensure successful app launches.
- identify and address platform-specific challenges encountered during the deployment of Flutter applications. They will learn to troubleshoot common issues and understand best practices for optimizing their applications for different environments.

By achieving these objectives, students will be well-prepared to develop, test, and deploy Flutter applications that function seamlessly across multiple platforms, leveraging Flutter's capabilities to create a cohesive user experience.

11.2 INTRODUCTION

Flutter is a powerful framework that enables developers to create applications that work across multiple platforms, such as iOS, Android, web, and desktop, all from a single codebase. This chapter will explore Flutter’s cross-platform capabilities, how to write platform-specific code, techniques for debugging and testing apps on different platforms, and the steps involved in deploying your app to Google Play Store, Apple App Store, and the web.

11.3 UNDERSTANDING FLUTTER'S CROSS-PLATFORM CAPABILITIES

Flutter offers an efficient way to develop apps that run seamlessly across multiple platforms, including mobile (iOS, Android), web, and desktop (macOS, Windows, Linux). The benefits of using Flutter for cross-platform development include:

- **Single Codebase**: You write code once, and Flutter compiles it for multiple platforms.
- **Hot Reload**: Allows you to see changes instantly without losing the app's state.
- **Performance**: Flutter uses its rendering engine, Skia, providing native performance across platforms.
- **Unified UI**: Flutter allows you to maintain a consistent look and feel across devices.

Supported Platforms

- **Mobile**: iOS and Android are fully supported platforms.
- **Web**: Flutter allows you to run the same code on browsers with minimal changes.
- **Desktop**: You can compile Flutter applications to run on macOS, Windows, and Linux, making it suitable for desktop applications.

11.4 PLATFORM-SPECIFIC CODE AND PLUGINS

While Flutter offers a unified codebase, there are cases where you need platform-specific functionality. Flutter allows you to write platform-specific code or use existing plugins to access native features of each platform.

Platform Channels

- **Platform Channels**: This is a mechanism provided by Flutter to communicate between the Dart code and the native code (Java/Kotlin for Android, Swift/Objective-C for iOS).

Example: Calling native Android or iOS APIs from Flutter using Platform Channels:

  - Android code: Use Java/Kotlin to implement Android-specific functionality.
  - iOS code: Use Swift/Objective-C to interact with iOS-specific APIs.

Plugins

- **Pre-built Plugins**: Flutter has a large collection of community-contributed and official plugins available for adding platform-specific features like camera access, GPS, and push notifications.
- **Custom Plugins**: You can write your own plugins if the functionality you need isn't available.

Example:

```
import 'package:flutter/foundation.dart';
import 'package:flutter/services.dart';

class BatteryLevel {
  static const platform = MethodChannel('samples.flutter.dev/battery');

  Future<String> getBatteryLevel() async {
    try {
      final int result = await platform.invokeMethod('getBatteryLevel');
      return 'Battery level is $result%.';
    } catch (e) {
      return 'Failed to get battery level.';
    }
  }
}
```

11.5 DEBUGGING AND TESTING APPS ON DIFFERENT PLATFORMS

Debugging and testing are critical to ensure that your app works seamlessly across all platforms. Flutter provides several tools and strategies to make this process easier.

Debugging Tools

- **Flutter DevTools**: A suite of performance and debugging tools included in Flutter to help you diagnose and fix issues.
- **Platform-Specific Debugging**: Use Xcode for iOS and Android Studio for Android to debug platform-specific issues.

Testing Strategies

- **Unit Testing**: Test individual units of code to ensure they function as expected.
- **Widget Testing**: Ensure the UI behaves as expected by testing individual widgets.

- **Integration Testing**: Test your app's entire workflow, from start to finish, on real devices or simulators.

Platform-Specific Testing

- **iOS**: Use **Xcode's simulators** to test your app on different iPhone models.
- **Android**: Use **Android Studio's emulators** to test across various Android devices.
- **Web**: Test the app on different browsers and screen sizes using browser tools.
- **Desktop**: Test on different OS versions (Windows, macOS, Linux) by building and running the app natively.

11.6 PREPARING YOUR APP FOR DEPLOYMENT

Once your app is ready, the next step is deploying it to the appropriate platform(s). This process involves preparing the app for distribution, configuring store settings, and meeting the specific requirements of each platform.

Deploying to the Google Play Store

Steps for Android deployment:

1. **Create an APK or AAB**: Build the release version of your Android app using Flutter's build commands.
   ```
   flutter build apk --release
   flutter build appbundle --release  # Recommended for Play Store
   ```
2. **Sign the APK**: Before distributing your app, it must be signed with a release key.
3. **Prepare Store Listing**: Fill in all required fields like app name, description, and images in the Play Console.
4. **Upload APK/AAB**: Upload the signed APK/AAB to Google Play Console.

Deploying to the Apple App Store

Steps for iOS deployment:

1. **Build iOS Release**: Build the release version of your iOS app using Flutter.
   ```
   flutter build ios –release
   ```
2. **Set up Xcode**: Open the project in Xcode and ensure it meets all the App Store requirements.

3. **Sign and Archive**: Use Xcode to archive and sign your app.
4. **Upload via App Store Connect**: Submit the app using Xcode or Transporter to App Store Connect.

Deploying to the Web

Steps for web deployment:

1. **Build for Web**: Build the release version of your web app.
   ```
   flutter build web
   ```
2. **Host the App**: Deploy the built web files to a web hosting service like Firebase Hosting, AWS, or GitHub Pages.

Deploying to Desktop

Steps for desktop deployment:

1. **Build for Desktop**: Build the release version for the desired desktop platform.
   ```
   flutter build windows   # for Windows
   flutter build macos     # for macOS
   flutter build linux     # for Linux
   ```
2. **Distribute the App**: Package and distribute the app based on the platform requirements (e.g., MSI for Windows, DMG for macOS).

Check Your Progress

1. Which of the following platforms does Flutter support for app development?
   A) Only Android
   B) Android and iOS
   C) Android, iOS, Web, and Desktop
   D) Only Web
2. What is the purpose of Platform Channels in Flutter?
   A) To communicate between Dart and HTML
   B) To access native platform APIs
   C) To compile Dart code into native code
   D) To manage app state
3. Which command is used to build a release version of a Flutter app for Android?
   A) flutter build apk
   B) flutter create apk

C) flutter deploy android

D) flutter run android

4. Which testing strategy focuses on the functionality of individual UI components in Flutter?

   A) Unit Testing

   B) Widget Testing

   C) Integration Testing

   D) Performance Testing

5. In Flutter, which command can be used to build the web version of an application?

   A) flutter build web

   B) flutter run web

   C) flutter compile web

   D) flutter deploy web

11.7 LET US SUM UP

In this chapter, we explored the capabilities of Flutter as a cross-platform framework, handling platform-specific code, and debugging and testing on multiple platforms. Finally, we discussed how to prepare and deploy Flutter apps for Android, iOS, the web, and desktop platforms. Mastering these skills will allow you to build and distribute apps efficiently across multiple ecosystems.

11.8 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-C) Android, iOS, Web, and Desktop

2-B) To access native platform APIs

3-A) flutter build apk

4-B) Widget Testing

5- A) flutter build web

11.9 FURTHER READING

- https://flutter.dev/multi-platform

- https://docs.flutter.dev/deployment/android
- https://docs.flutter.dev/deployment/web
- https://docs.flutter.dev/deployment

11.10 ASSIGNMENT

1. Explain the benefits of using Flutter for cross-platform development. How does it differ from traditional platform-specific development approaches (e.g., native Android or iOS)?
2. Describe the role of platform channels in Flutter. Provide an example where platform-specific code is necessary in a multi-platform application and explain how you would implement it.
3. Outline the steps involved in preparing and deploying a Flutter application to both the Google Play Store and the Apple App Store. What are the key differences in the deployment process for these two platforms?
4. What are the different testing strategies available in Flutter? Discuss the importance of widget testing and integration testing when building cross-platform applications.
5. You are tasked with deploying a Flutter app to the web. Explain the steps required to build the app for the web and list at least two hosting options. What challenges might arise during this process?

11.11 ACTIVITIES

1. Build a simple app that uses platform-specific features such as accessing the device's camera on Android and iOS.
2. Deploy your app to both the Google Play Store and Apple App Store.
3. Create a basic web version of your app and deploy it using Firebase Hosting or any other web hosting service.

Unit-12: Performance Optimization in Flutter

12

Unit Structure

12.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Explain why performance optimization is critical in mobile applications.
- Identify the factors that contribute to a positive user experience and how performance impacts user engagement.
- Describe the concept of widget rebuilding in Flutter and its implications for performance.
- Demonstrate the use of const constructors to create immutable widgets.
- Apply the ValueListenableBuilder to update specific parts of the UI without unnecessary rebuilds.
- Utilize scoped setState calls to limit widget rebuilds to only those that require updates
- Differentiate between various types of Flutter widgets and their performance characteristics.
- Implement ListView.builder for rendering large lists efficiently.
- Use IndexedStack to switch between widgets while maintaining their state.
- Analyze the structure of the build method to minimize its complexity.
- Apply techniques to offload heavy computations from the build method using compute() or asynchronous calls.
- Employ RepaintBoundary to isolate parts of the widget tree that require frequent updates.
- Identify best practices for creating smooth animations and rendering in Flutter.
- Utilize Flutter’s animation library components, such as AnimatedBuilder and AnimatedContainer.
- Monitor frame rates to ensure animations run at 60 fps and diagnose frame drops using performance tools.
- Familiarize with the various tools available in Flutter DevTools for performance analysis.
- Interpret the data provided by the Flutter Inspector to identify inefficiencies in the widget tree.
- Use the performance overlay and timeline view to assess frame rendering times and identify bottlenecks.

- Conduct memory profiling to detect memory leaks and optimize memory usage.
- Implement asynchronous programming to prevent blocking the UI thread during network calls.
- Develop caching strategies to minimize redundant API calls and improve data retrieval times.
- Apply best practices for batching API requests to reduce network overhead.
- Choose efficient data formats and consider data compression techniques to enhance network performance.

12.2 INTRODUCTION

Performance is a critical factor in mobile app development, as it greatly influences user satisfaction and engagement. Flutter, with its rich set of features and flexibility, offers various ways to optimize application performance. In this chapter, we will explore essential techniques for optimizing Flutter apps, focusing on minimizing rebuilds, enhancing animations, profiling performance, and optimizing network and API calls. By the end of this chapter, you will have a solid understanding of how to create high-performance Flutter applications.

12.3 TECHNIQUES FOR OPTIMIZING FLUTTER APPS

Minimizing Rebuilds

Unnecessary widget rebuilds can lead to performance bottlenecks in Flutter applications. Flutter rebuilds a widget whenever its parent widget is rebuilt. To minimize this:

- **Use const Widgets**: Marking a widget as const tells Flutter that it will not change, preventing it from rebuilding unnecessarily.
  ```
  const Text('Hello, World!');  // This widget won't rebuild
  ```
- **ValueListenableBuilder**: This widget listens to a ValueNotifier and only rebuilds when the value changes. This is an efficient way to update specific parts of the UI.

```
ValueListenableBuilder<int>(
  valueListenable: myValueNotifier,
  builder: (context, value, child) {
    return Text('Value: $value');
  },
);
```

Scoped setState Calls: Limit the scope of setState() to specific widgets that need to be rebuilt rather than calling it at a higher level in the widget tree.

Efficient Widget Usage

Choosing the right widget can significantly improve performance:

- **Use ListView.builder**: For lists with many items, ListView.builder builds only the visible items, enhancing performance by reusing widgets.
- **IndexedStack**: When switching between multiple widgets, use IndexedStack to maintain the state of each widget while only displaying the active one.

Optimize Build Method

The build method is called frequently, so minimizing its complexity is essential:

- **Avoid Heavy Computation**: Move heavy computations out of the build method. Use compute() to run tasks in a separate isolate if needed.
- **Use RepaintBoundary**: This widget can help improve performance by isolating parts of the widget tree that need frequent updates from the rest of the UI.

12.4 BEST PRACTICES FOR SMOOTH ANIMATIONS AND RENDERING

Animations play a vital role in user experience. To ensure smooth animations:

Leverage the Animation Library

Utilize Flutter’s animation library, which includes:

- **AnimatedBuilder**: This widget allows you to efficiently animate properties without having to write boilerplate code.

- **AnimatedContainer**: Use this for simple animations that involve size, padding, color, etc. It automatically animates changes to these properties.

Optimize Animation Performance

- **Avoid Complex Calculations**: Keep animations lightweight by avoiding complex calculations in the animation loop.
- **Profile Animations**: Use the performance overlay to identify dropped frames and performance issues during animations.

Manage Frame Rates

- Ensure animations are running at 60 frames per second (fps). Profiling your app can help identify and address frame drops, ensuring a fluid experience.

12.5 PROFILING AND ANALYZING APP PERFORMANCE USING FLUTTER DEVTOOLS

Profiling is essential for identifying performance bottlenecks. Flutter provides **DevTools**, a suite of tools for performance analysis:

Flutter Inspector

The Flutter Inspector allows developers to visualize the widget tree, enabling them to identify inefficiencies and unnecessary rebuilds.

Performance Overlay

Enabling the performance overlay displays information about the frame rendering times, helping to identify if your app is dropping frames.

Timeline View

The timeline view provides detailed insights into frame rendering times, build durations, and event handling. This can help developers track where time is being spent in their app.

Memory Profiling

Memory profiling tools in Flutter DevTools can help identify memory leaks and optimize memory usage. Keep track of object allocation and deallocation to ensure efficient memory management.

12.6 OPTIMIZING NETWORK AND API CALLS FOR BETTER PERFORMANCE

Network calls can greatly affect application performance. Here’s how to optimize them:

Use Asynchronous Calls

Always perform network calls asynchronously to prevent blocking the UI thread. Use async/await for non-blocking calls:

```
Future<void> fetchData() async {
  final response = await http.get(Uri.parse('https://example.com/data'));
  // Process response
}
```

Implement Caching Strategies

Implement caching strategies to reduce redundant API calls. Use packages like shared_preferences or hive for local storage. This can significantly improve performance by serving cached data instead of making network requests.

Batch API Requests

If your application requires multiple API calls, batch them together if possible. This reduces the number of network requests and can lead to improved performance.

Efficient Data Formats

Choose lightweight data formats like JSON instead of XML. Consider compressing the data sent over the network to reduce load times.

Check Your Progress

1. What is the primary benefit of using const constructors for widgets in Flutter?

A) They allow for animations to run smoother.

B) They prevent unnecessary rebuilds of immutable widgets.

C) They reduce the size of the app.

D) They enable the use of platform-specific code.

2. Which widget is best suited for rendering a large list of items efficiently in Flutter?

A) ListView   B) ListView.builder   C) Column   D) GridView

3. What tool can you use in Flutter to profile the performance of your application?

A) Flutter Inspector   B) Android Studio   C) Visual Studio Code   D) DartPad

4. Which of the following is a technique to reduce frame drops during animations in Flutter?

A) Using heavy computation in the animation loop.

B) Ensuring animations run at 60 frames per second (fps).

C) Ignoring the performance overlay.

D) Using the same widget for all animations.

5. What is one method to optimize network calls in a Flutter application?

A) Making all API calls synchronous.

B) Using caching strategies to reduce redundant requests.

C) Avoiding asynchronous programming altogether.

D) Increasing the number of API calls made.

12.7 LET US SUM UP

Performance optimization is a crucial aspect of Flutter app development. By employing techniques to minimize rebuilds, enhance animations, and efficiently manage network calls, developers can create responsive and high-performing applications. Regular profiling using Flutter DevTools will ensure that any performance bottlenecks are identified and resolved early in the development process. With these strategies, developers can provide an exceptional user experience that meets the demands of modern mobile applications.

12.8 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-B) They prevent unnecessary rebuilds of immutable widgets.

2- B) ListView.builder   3-A) Flutter Inspector

4-B) Ensuring animations run at 60 frames per second (fps).

5-B) Using caching strategies to reduce redundant requests.

12.9 FURTHER READING

- https://docs.flutter.dev/perf/best-practices
- https://docs.flutter.dev/perf
- https://docs.flutter.dev/perf/rendering-performance
- https://docs.flutter.dev/perf/ui-performance

12.10 ASSIGNMENT

1. Explain the concept of widget rebuilding in Flutter. How does it impact the performance of an application? Provide examples of when a widget might need to rebuild.
2. Discuss the best practices for creating smooth animations in Flutter. What techniques can you use to ensure that your animations run at 60 frames per second?
3. Describe how you would use Flutter DevTools to profile the performance of an application. What specific metrics or tools within DevTools would you focus on, and why?
4. Consider a scenario where your Flutter application makes multiple network calls to an API. What strategies can you implement to optimize these calls for better performance? Discuss the advantages of each strategy.
5. Reflect on the role of the build method in Flutter. What are some common pitfalls that developers might encounter when writing the build method, and how can they mitigate these issues to improve performance?

12.11 ACTIVITIES

Activity: Widget Optimization Challenge

Objective: To practice identifying and optimizing widget rebuilds in a Flutter application.

Instructions:

1. **Create a Sample Application**: Build a simple Flutter application that includes a list of items (e.g., a list of names or products). Use ListView to display the items.

2. **Identify Rebuilds**: Modify the application to introduce a stateful widget that alters some of the items in the list. Implement a setState() method that updates the list based on user interaction (e.g., adding or removing items).
3. **Profile Performance**: Use Flutter DevTools to profile the performance of the application. Pay attention to the performance overlay and look for any dropped frames.
4. **Optimize the Code**:
   - Identify areas where unnecessary widget rebuilds occur and refactor the code to use const widgets or ValueListenableBuilder where applicable.
   - Re-test the performance after making changes.
5. **Presentation**: Prepare a short presentation (5-10 minutes) to share your findings, focusing on how you identified the performance issues and what optimizations you implemented.

Unit-13: Custom Widgets and Plugins

13

Unit Structure

13.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Create custom **Stateless** and **Stateful** widgets tailored to specific design requirements.
- Leverage **composition** to combine existing widgets and build complex UI components.
- Grasp the lifecycle of a **StatefulWidget** and manage state efficiently using initState(), setState(), and dispose() methods.
- Implement stateful behaviors in widgets to handle dynamic content and interactions.
- Understand the architecture of **Flutter plugins** and the role of **platform channels** for communication between Dart and native code.
- Build basic custom plugins to access platform-specific features (e.g., camera, battery status) on both Android and iOS.
- Implement and use **MethodChannels** for two-way communication between Flutter and platform-specific APIs.
- Differentiate between **MethodChannel**, **EventChannel**, and **BasicMessageChannel**, and know when to use each type of channel.
- Develop reusable Flutter plugins for personal projects or the community.
- Understand the process of packaging and publishing a plugin to **pub.dev**, making it available for use by other developers.
- Optimize custom widgets for reusability and performance.
- Ensure that plugins are cross-platform compatible and follow Flutter's guidelines for native code integration.

13.2 INTRODUCTION

In Flutter, everything is a widget. From buttons to complex layouts, all visual elements are widgets. However, as applications grow, the need for custom, reusable, and efficient widgets becomes essential. In this chapter, we'll dive into advanced techniques for building **custom widgets** and creating **plugins** to extend Flutter's functionality beyond what's available out of the box.

13.3 CUSTOM WIDGETS

Flutter provides a vast library of pre-built widgets that cover most common use cases. However, real-world applications often require bespoke visual elements and behaviors. Custom widgets offer flexibility, enabling developers to create reusable components with custom styling, animations, and interactions.

There are two main types of custom widgets in Flutter:

- **StatelessWidget**: A widget that doesn't change its state over time.
- **StatefulWidget**: A widget that maintains some state, which can change and trigger re-renders.

Building Custom Stateless Widgets

Let's start by creating a custom **StatelessWidget**. This type of widget is best suited for static content or content that doesn't need to change during the app's lifecycle.

```
class CustomButton extends StatelessWidget {
  final String label;
  final VoidCallback onPressed;

  const CustomButton({required this.label, required this.onPressed});

  @override
  Widget build(BuildContext context) {
    return ElevatedButton(
      onPressed: onPressed,
      child: Text(label),
      style: ElevatedButton.styleFrom(
        primary: Colors.blue, // Custom color
        padding: EdgeInsets.symmetric(horizontal: 20, vertical: 15),
        textStyle: TextStyle(fontSize: 18),
      ),
    );
  }
}
```

In this example, CustomButton takes a label and an onPressed callback. It allows developers to easily reuse this widget throughout their application.

Building Custom Stateful Widgets

Now, let's create a custom **StatefulWidget**. This is useful for widgets that need to manage internal state, such as form inputs or toggle switches.

```
class ToggleSwitch extends StatefulWidget {
  final bool initialValue;
  final ValueChanged<bool> onChanged;

  const ToggleSwitch({required this.initialValue, required this.onChanged});

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

class _ToggleSwitchState extends State<ToggleSwitch> {
  late bool isActive;

  @override
  void initState() {
    super.initState();
    isActive = widget.initialValue;
  }

  @override
  Widget build(BuildContext context) {
    return Switch(
      value: isActive,
      onChanged: (value) {
        setState(() {
          isActive = value;
        });
        widget.onChanged(value);
      },
    );
  }
}
```

Here, ToggleSwitch keeps track of its internal state (isActive). It can be reused anywhere that needs a custom toggle switch with additional behaviors, such as callbacks when the switch is toggled.

13.4 CUSTOMIZING WIDGETS USING COMPOSITION

Instead of subclassing a widget, Flutter encourages **composition** — creating new widgets by combining existing ones. This technique is more flexible, reusable, and fits well within Flutter's declarative UI framework.

For example, you can compose widgets to build a more complex UI component:

```
class UserProfileCard extends StatelessWidget {
  final String userName;
  final String imageUrl;

  const UserProfileCard({required this.userName, required this.imageUrl});

  @override
  Widget build(BuildContext context) {
    return Card(
      elevation: 4,
      child: Column(
        children: [
          Image.network(imageUrl, width: 100, height: 100),
          SizedBox(height: 10),
          Text(userName, style: TextStyle(fontSize: 20, fontWeight: FontWeight.bold)),
          SizedBox(height: 10),
        ],
      ),
    );
  }
}
```

This UserProfileCard combines several Flutter widgets (like Card, Column, and Text) to form a reusable UI component.

13.5 UNDERSTANDING WIDGET LIFECYCLE

For advanced Flutter development, it's essential to understand the **widget lifecycle**, especially for **StatefulWidgets**.

The key lifecycle methods for StatefulWidget include:

- **initState()**: Called once when the widget is inserted into the widget tree. It's commonly used to initialize state.
- **build()**: Called every time the widget is rendered, based on the current state.
- **dispose()**: Called when the widget is removed from the tree permanently. This is the place to clean up resources like streams or controllers.

Example using lifecycle methods:

```
class TimerWidget extends StatefulWidget {
  @override
  _TimerWidgetState createState() => _TimerWidgetState();
}

class _TimerWidgetState extends State<TimerWidget> {
  late Timer _timer;
  int _seconds = 0;
  @override
  void initState() {
    super.initState();
    _timer = Timer.periodic(Duration(seconds: 1), (Timer t) {
      setState(() {
        _seconds++;
      });
    });
  }

  @override
  void dispose() {
    _timer.cancel();
    super.dispose();
  }

  @override
  Widget build(BuildContext context) {
    return Text('Seconds elapsed: $_seconds');
  }
}
```

This widget starts a timer in initState() and cleans it up in dispose() to avoid memory leaks.

13.6 CREATING PLUGINS IN FLUTTER

While Flutter's widget library covers most use cases, sometimes you need to interact with **platform-specific features** like the camera, GPS, or file system. Plugins provide a way to extend Flutter's functionality by writing **platform-specific code** for iOS and Android.

Plugin Architecture

Flutter plugins are structured in a way that allows communication between Dart and platform-specific code (Java/Kotlin for Android, Objective-C/Swift for iOS). This is achieved through **platform channels**.

A platform channel allows you to send messages between Dart and the host platform (Android or iOS) using a method call mechanism.

Key Concepts:

- **MethodChannel**: Used to communicate with native code.
- **EventChannel**: Used to listen to continuous streams of data (e.g., sensor data).
- **BasicMessageChannel**: Used for binary or string messages.

Creating a Basic Plugin

Here's a simplified example of how you might create a plugin that interacts with the native battery level on both Android and iOS.

```
class BatteryLevel {
  static const MethodChannel _channel = MethodChannel('battery');

  static Future<int> getBatteryLevel() async {
    final int batteryLevel = await _channel.invokeMethod('getBatteryLevel');
    return batteryLevel;
  }
}
```

Android Native Code (Kotlin):

```
class MainActivity: FlutterActivity() {
    private val CHANNEL = "battery"

    override fun configureFlutterEngine(flutterEngine: FlutterEngine) {
        super.configureFlutterEngine(flutterEngine)

        MethodChannel(flutterEngine.dartExecutor.binaryMessenger,
CHANNEL).setMethodCallHandler { call, result ->
            if (call.method == "getBatteryLevel") {
                val batteryLevel = getBatteryLevel()

                if (batteryLevel != -1) {
                    result.success(batteryLevel)
                } else {
                    result.error("UNAVAILABLE", "Battery level not available.", null)
                }
            } else {
                result.notImplemented()
            }
        }
    }

    private fun getBatteryLevel(): Int {
        val batteryLevel: Int
        val batteryIntent = ContextCompat.getSystemService(this,
BatteryManager::class.java)
```

```
        batteryLevel                                                         =
batteryIntent?.getIntProperty(BatteryManager.BATTERY_PROPERTY_CAPACITY)
?: -1
        return batteryLevel
    }
}
```

iOS Native Code (Swift):

```
import Flutter
import UIKit

public class SwiftBatteryPlugin: NSObject, FlutterPlugin {
  public static func register(with registrar: FlutterPluginRegistrar) {
    let channel = FlutterMethodChannel(name: "battery", binaryMessenger:
registrar.messenger())
    let instance = SwiftBatteryPlugin()
    registrar.addMethodCallDelegate(instance, channel: channel)
  }

  public func handle(_ call: FlutterMethodCall, result: @escaping FlutterResult) {
    if (call.method == "getBatteryLevel") {
      result(getBatteryLevel())
    } else {
      result(FlutterMethodNotImplemented)
    }
  }

  private func getBatteryLevel() -> Int {
    let device = UIDevice.current
    device.isBatteryMonitoringEnabled = true
    return Int(device.batteryLevel * 100)
  }
}
```

13.7 PUBLISHING AND USING PLUGINS

Once you've created your plugin, you can publish it to **pub.dev**, Flutter's official package repository, or use it locally in your project. Publishing a plugin makes it accessible to the Flutter community, and you can also benefit from contributions to improve your plugin.

To publish, you need to:

1. Add metadata (name, description, etc.) to your pubspec.yaml.
2. Ensure that your plugin works correctly on both Android and iOS.
3. Run flutter packages pub publish to upload the package.

Check Your Progress

1. What is the primary difference between a StatelessWidget and a StatefulWidget in Flutter?

   A) StatelessWidget does not have a build() method, while StatefulWidget does.

   B) StatelessWidget does not manage state, while StatefulWidget manages mutable state.

   C) StatelessWidget can change its state over time, while StatefulWidget cannot.

   D) StatefulWidget is used for performance optimization, while StatelessWidget is not.

2. Which lifecycle method in a StatefulWidget is used to perform one-time initialization tasks?

   A) dispose()

   B) build()

   C) initState()

   D) setState()

3. In Flutter, what is the best practice for creating complex UI components from smaller widgets?

   A) Extending a single widget class for all UI components.

   B) Using the setState() method extensively.

   C) Subclassing existing widgets for every new component.

   D) Composing existing widgets to build new complex widgets.

4. Which of the following Flutter plugins is designed to listen to continuous streams of data, like sensor input?

   A) MethodChannel

   B) EventChannel

   C) BasicMessageChannel

   D) PlatformChannel

5. What is the purpose of the MethodChannel in Flutter plugins?

   A) To communicate between Flutter and platform-specific code using method calls.

   B) To handle continuous data streams between Flutter and native code.

   C) To send binary data between Flutter and platform code.

   D) To define widget lifecycle methods.

13.8 LET US SUM UP

Mastering custom widgets and plugins in Flutter allows developers to build highly customized, performant applications that cater to any design or functionality requirement. Custom widgets enable flexibility and reusability, while plugins extend the power of Flutter to platform-specific features, bridging the gap between

13.9 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-B, 2-C 3-D 4-B 5-A

13.10 FURTHER READING

- https://docs.flutter.dev/packages-and-plugins/developing-packages
- https://docs.flutter.dev/ui/widgets
- https://docs.flutterflow.io/concepts/custom-code/custom-widgets/
- https://docs.flutter.dev/packages-and-plugins

13.11 ASSIGNMENT

- What are the benefits of using custom widgets in Flutter, and how can you ensure that they are reusable across different parts of an application?
- When would you prefer creating a custom StatefulWidget over a StatelessWidget? Provide an example scenario.
- Explain the concept of widget composition in Flutter. How does it differ from subclassing? Provide a real-world example of how composition improves flexibility and maintainability in app development.
- Describe the lifecycle of a StatefulWidget in Flutter. What happens if you forget to override the dispose() method in a widget that manages resources such as streams or timers?
- Imagine you need to access the device's camera functionality for an app. What steps would you follow to create a custom Flutter plugin that interacts with the platform's native camera API?
- Why is it important to ensure that plugins are compatible with both Android and iOS? What are some potential challenges developers face in maintaining cross-platform compatibility?

- Discuss how platform channels work in Flutter. How would you use a MethodChannel to invoke a native function that retrieves the device's GPS coordinates? What considerations must you keep in mind when communicating between Dart and native code?

13.12 ACTIVITIES

- Create a **Battery Level Display** widget that retrieves and shows the device's current battery level using platform-specific native code.

Unit-14: Testing in Flutter 14

Unit Structure

14.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Distinguish between unit tests, widget tests, and integration tests, and understand the purpose of each type.
- Know when to apply each type of test to ensure comprehensive coverage for your Flutter applications.
- Write unit tests to verify the correctness of individual functions or classes in isolation.
- Use the flutter_test package to test logic and edge cases in your code.
- Run unit tests and interpret the results to improve code reliability.
- Write widget tests to validate the UI and interactions of individual widgets.
- Utilize the WidgetTester class to simulate user interactions like taps and verify that widgets behave as expected.
- Write integration tests to simulate real-world user behavior and interactions across multiple screens and widgets.
- Use the integration_test package to automate end-to-end testing on devices and emulators.
- Set up an automated CI pipeline to run tests on every commit or pull request, ensuring that your app's code remains stable as it evolves.
- Integrate testing into platforms like GitHub Actions, CircleCI, or other CI services.
- Apply best practices for writing clear, maintainable tests across all levels (unit, widget, and integration).
- Mock external dependencies when necessary to isolate tests and avoid reliance on real-world services.
- Optimize test performance to ensure that your testing process is efficient and scalable.

14.2 INTRODUCTION

Testing is an essential aspect of software development that ensures the reliability, correctness, and performance of applications. In Flutter, testing is made simple and comprehensive, providing several tools and methodologies to validate the functionality of your app at multiple levels. This chapter will guide you through the different types of testing in Flutter and demonstrate best practices to maintain a robust, high-quality application.

14.3 TYPES OF TESTING IN FLUTTER

Testing in Flutter can be categorized into three main types:

- **Unit Testing**: Testing individual units of code (e.g., a single function or class) in isolation.
- **Widget Testing**: Testing individual widgets to ensure they render correctly and behave as expected in different scenarios.
- **Integration Testing**: Testing the app as a whole to simulate user behavior and interactions across multiple widgets and screens.

Each type serves a different purpose in ensuring your app's functionality, reliability, and performance.

14.4 UNIT TESTING IN FLUTTER

What is Unit Testing?

Unit tests validate the functionality of individual units of code, such as methods or classes, ensuring that they behave correctly under various conditions. These tests are fast, isolated, and form the foundation of your test suite, allowing you to verify the smallest parts of your app's logic.

Writing Unit Tests

Flutter provides the flutter_test package, which includes tools for writing unit tests. Here's a basic example of a unit test for a function that adds two numbers:

```
// math_functions.dart
int add(int a, int b) {
  return a + b;
}

// math_functions_test.dart
import 'package:flutter_test/flutter_test.dart';
import 'math_functions.dart';
```

```
void main() {
  test('Addition of two numbers', () {
    expect(add(2, 3), 5);
  });
}
```

Running Unit Tests

To run the unit test, use the following command in the terminal:

```
flutter test
```

This will run all the tests in the test directory and output the results in the console.

Best Practices for Unit Testing

- **Isolate Logic**: Ensure that each test focuses on a small piece of functionality. Avoid dependencies on external systems like APIs or databases.
- **Mock External Dependencies**: Use mocking frameworks like mockito to replace real objects with mock ones, simulating their behavior without relying on the actual implementation.
- **Test Edge Cases**: Write tests for typical cases and also test edge cases (e.g., empty inputs, null values, etc.) to improve coverage.

14.5 WIDGET TESTING IN FLUTTER

What is Widget Testing?

Widget tests (also called component tests) check the UI and interactions of individual widgets. These tests ensure that a widget renders correctly and responds to user actions (like tapping a button or entering text) as expected. Widget testing provides a middle ground between unit testing and integration testing.

Writing Widget Tests

Here's an example of a widget test for a simple counter app:

```
// counter.dart
class Counter extends StatefulWidget {
  @override
  _CounterState createState() => _CounterState();
}

class _CounterState extends State<Counter> {
  int _count = 0;

  void _increment() {
    setState(() {
      _count++;
    });
  }

  @override
  Widget build(BuildContext context) {
    return Column(
      children: [
        Text('Count: $_count'),
        ElevatedButton(
          onPressed: _increment,
          child: Text('Increment'),
        ),
      ],
    );
  }
}


// counter_test.dart
import 'package:flutter/material.dart';
import 'package:flutter_test/flutter_test.dart';
import 'counter.dart';

void main() {
  testWidgets('Counter increments when button is pressed', (WidgetTester tester)
async {
    // Build the widget
    await tester.pumpWidget(MaterialApp(home: Counter()));

    // Verify initial counter value is 0
    expect(find.text('Count: 0'), findsOneWidget);

    // Tap the increment button
    await tester.tap(find.byType(ElevatedButton));
    await tester.pump(); // Rebuild the widget

    // Verify the counter value has incremented
    expect(find.text('Count: 1'), findsOneWidget);
  });
}
```

Understanding the WidgetTester

The WidgetTester object simulates user interactions and allows you to perform actions like:

- **pumpWidget()**: Builds the widget and places it in the test environment.
- **tap()**: Simulates tapping on a widget (like a button).
- **pump()**: Rebuilds the widget after a state change.

Running Widget Tests

Run widget tests using the same command as unit tests:

```
flutter test
```

Best Practices for Widget Testing

- **Test UI and Behavior**: Ensure that widgets not only display correctly but also behave as expected when interacted with.
- **Use Keys for Finding Widgets**: Assign keys to widgets when necessary to make them easier to find in tests, especially for widgets with similar content.
- **Simulate Different Scenarios**: Test how the widget responds to different conditions, such as varying screen sizes or input values.

14.6 INTEGRATION TESTING IN FLUTTER

What is Integration Testing?

Integration tests validate the app's behavior as a whole, simulating real-world user interactions. These tests cover multiple widgets and screens, ensuring that they work together correctly.

Integration tests are often more complex and slower to run than unit or widget tests, but they are crucial for validating the overall app experience.

Writing Integration Tests

Flutter provides the integration_test package for writing and running integration tests. Here’s an example of an integration test that tests the same counter app:

```
// integration_test/counter_integration_test.dart
import 'package:flutter_test/flutter_test.dart';
import 'package:integration_test/integration_test.dart';
import 'package:flutter/material.dart';
import 'package:my_app/counter.dart';

void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized();

  testWidgets('Counter increments in integration test', (WidgetTester tester) async {
    await tester.pumpWidget(MaterialApp(home: Counter()));

    // Verify initial count
    expect(find.text('Count: 0'), findsOneWidget);

    // Simulate tap
    await tester.tap(find.byType(ElevatedButton));
    await tester.pump();

    // Verify updated count
    expect(find.text('Count: 1'), findsOneWidget);
  });
}
```

Running Integration Tests

Integration tests are executed differently than unit and widget tests. You'll need to use the following command to run integration tests on an actual device or emulator:

flutter test integration_test/counter_integration_test.dart

Alternatively, you can use the following command to run all tests in the integration_test folder:

flutter test integration_test

Best Practices for Integration Testing

- **Test Critical User Flows**: Focus on testing critical user journeys (like logging in, making a purchase, etc.), where failures would significantly impact the app's usability.
- **Use Mock Data**: When possible, mock external services (like APIs) to isolate the test environment and avoid relying on external systems.
- **Limit Integration Tests**: Since they can be slow, limit the number of integration tests by focusing only on the most important flows.

14.7 AUTOMATED TESTING WORKFLOW IN FLUTTER

Test Automation and Continuous Integration (CI)

Automating tests and integrating them into a **Continuous Integration (CI)** pipeline ensures that code is consistently tested as it evolves. Popular CI platforms like GitHub Actions, CircleCI, and Travis CI support Flutter testing.

Here's a simple configuration for **GitHub Actions** to automate testing:

```yaml
name: Flutter Test

on: [push, pull_request]


jobs:
  test:
    runs-on: ubuntu-latest


    steps:
    - uses: actions/checkout@v2
    - uses: subosito/flutter-action@v2
      with:
        flutter-version: '2.5.0'
    - run: flutter pub get
    - run: flutter test
```

This YAML file defines a CI workflow that runs Flutter tests automatically whenever you push code or submit a pull request.

Continuous Integration Best Practices

- **Run Tests on Every Commit**: Ensure that your test suite is automatically triggered for every commit or pull request.
- **Monitor Test Performance**: Pay attention to how long tests take to run. Long-running tests might slow down your development process.
- **Fail Early, Fail Fast**: Ensure that failing tests provide quick feedback, allowing you to fix bugs or issues before they propagate.

Check Your Progress

1. What is the primary purpose of unit testing in Flutter?

   A) To test the entire user interface

   B) To validate individual units of code in isolation

   C) To test interactions between different widgets

   D) To simulate real-world user behavior across screens

2. Which tool is used for simulating user interactions in Flutter widget tests?

   A) WidgetBuilder

   B) IntegrationTester

   C) UnitTester

   D) WidgetTester

3. In integration testing, what does the pump() method do in the WidgetTester class?

   A) Simulates user interaction with the widget

   B) Verifies the test result

   C) Rebuilds the widget after a state change

   D) Initializes the widget tree

4. What is the recommended practice for testing widgets that depend on external services like APIs?

   A) Avoid testing those widgets altogether

   B) Use the real service during the test

   C) Use mock services or mock data

   D) Skip testing the widgets for external services

5. Which of the following commands runs all the tests in the test directory of a Flutter project?

   A) flutter run test

   B) flutter analyze

   C) flutter test

   D) flutter check

6. Integration tests in Flutter are faster to run than unit tests. (True/False)

7. In Flutter, MethodChannel is primarily used for communicating between Dart and native platform code. (True/False)

14.8 LET US SUM UP

Testing is a vital part of Flutter app development, ensuring that your code is correct, stable, and ready for production. By integrating **unit tests**, **widget tests**, and **integration tests** into your workflow, you can build a strong test suite that validates the functionality of individual components, the behavior of widgets, and the app's performance as a whole.

Additionally, by automating your tests using a continuous integration pipeline, you can ensure that your app remains reliable as it grows and evolves. The key to successful testing lies in balancing test types, focusing on critical flows, and maintaining a fast and efficient test suite.

With the right testing strategy in place, you can ensure that your Flutter apps are bug-free, performant, and ready to provide a seamless user experience

14.9 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1-B) To validate individual units of code in isolation

2-D) WidgetTester

3-C) Rebuilds the widget after a state change

4- C) Use mock services or mock data

5-C) flutter test   6-False   7-True

14.10 FURTHER READING

- https://docs.flutter.dev/testing/overview
- https://docs.flutter.dev/cookbook/testing/unit/introduction
- https://docs.flutter.dev/cookbook/testing/integration/introduction
- https://docs.flutter.dev/cookbook/testing/widget/introduction

14.11 ASSIGNMENT

1. What is the difference between unit testing and widget testing in Flutter? Can you provide an example where each type would be most appropriate?
2. Explain how WidgetTester is used to simulate user interactions in a widget test. Why is it important to use pump() after certain actions in widget tests?

3. When should you consider using integration tests in a Flutter app? Provide an example of a scenario where integration testing would be necessary.
4. Describe the role of mock objects in testing Flutter applications. How do they improve the reliability of tests, particularly when dealing with external dependencies like APIs?
5. What are some challenges you might face when writing integration tests, and how can you overcome them to ensure your tests remain efficient and effective?
6. Why is it important to include testing as part of your Continuous Integration (CI) workflow? How does this help in maintaining the quality of your Flutter application over time?
7. Can you think of a real-world scenario where testing only at the unit level would be insufficient? What other levels of testing should be added to ensure the app works as expected?

14.12 ACTIVITIES

1. Practice writing unit and widget tests for a basic Flutter application.
2. Mocking External Services
3. Setting up automated testing in a CI environment.

Unit-15: Flutter for the Web and Desktop

15

Unit Structure

15.1 LEARNING OBJECTIVE

After studying this unit student should be able to:

- Identify the advantages of using Flutter to build applications for web and desktop platforms, including a single codebase and rich UI components.
- Configure the Flutter SDK for web and desktop development, ensuring all necessary tools and settings are in place for creating cross-platform applications.
- Build a basic Flutter web application and run it in a web browser, demonstrating knowledge of Flutter's web capabilities.
- Use MediaQuery, LayoutBuilder, and responsive widgets to create adaptable user interfaces that work seamlessly across different screen sizes and orientations on the web.
- Implement named routes and manage navigation in Flutter web applications, enhancing user experience with intuitive page transitions.
- Create a Flutter desktop application for Windows, macOS, or Linux, and run it in a native environment.
- Incorporate desktop-specific functionalities such as window management and native API access to enhance the user experience in desktop applications.
- Implement performance optimization strategies and accessibility features to ensure applications run efficiently and are usable by a diverse audience.
- Prepare and deploy Flutter web applications to web servers and package desktop applications for distribution on different platforms.
- Recognize the importance of keeping up with the latest updates and best practices in the Flutter ecosystem to continuously improve application development skills.

15.2 INTRODUCTION

Flutter, primarily known for building mobile applications, has extended its capabilities to support web and desktop platforms, enabling developers to create cross-platform applications from a single codebase. This chapter explores how to leverage Flutter for web and desktop development, covering the key concepts, tools, and best practices needed to build responsive and performant applications.

15.3 FLUTTER FOR WEB AND DESKTOP

Why Flutter for Web and Desktop?

Flutter provides several advantages for building web and desktop applications:

- **Single Codebase**: Write once, run anywhere. Flutter allows you to maintain a single codebase for multiple platforms, reducing development and maintenance efforts.
- **Rich UI Components**: Flutter's rich set of pre-built widgets enables the creation of visually appealing applications with a consistent look and feel across platforms.
- **Hot Reload**: The hot reload feature in Flutter accelerates the development process by allowing developers to see changes instantly without restarting the application.
- **Performance**: Flutter compiles to native code, providing high performance and responsiveness, even on web and desktop platforms.

Overview of Flutter for Web and Desktop

Flutter supports building applications for the following platforms:

- **Web**: Targeting browsers, Flutter for web allows developers to create responsive web applications that run on any device with a modern web browser.
- **Desktop**: Flutter for desktop enables building native desktop applications for Windows, macOS, and Linux, providing a familiar environment for users.

15.4 GETTING STARTED WITH FLUTTER FOR WEB

Setting Up the Environment

To get started with Flutter for web, you need to ensure you have the following installed:

1. **Flutter SDK**: Install the latest version of Flutter from the official website.
2. **Web Development Tools**: Make sure you have a compatible browser (Chrome is recommended) for testing your web applications.
3. **Editor**: Use an editor like Visual Studio Code or Android Studio with the Flutter and Dart plugins.

Creating a Flutter Web Project

To create a new Flutter web project, use the following command:

```
flutter create my_web_app
```

Navigate to the project directory:

```
cd my_web_app
```

To run the project in a web browser, execute:

```
flutter run -d chrome
```

Building Responsive Layouts

Creating responsive layouts is crucial for web applications. Flutter offers several ways to achieve this:

- **MediaQuery**: Use MediaQuery to get the screen size and orientation to adjust your layout dynamically.
- **LayoutBuilder**: Utilize LayoutBuilder to build widgets based on the available space.
- **Responsive Widgets**: Consider using responsive widgets like Flexible, Expanded, and AspectRatio to create adaptive layouts.

Example of a Responsive Layout:

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

class ResponsiveLayout extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return LayoutBuilder(
      builder: (context, constraints) {
        if (constraints.maxWidth < 600) {
          return Column(
            children: [
              Text('Small Screen'),
              // Add mobile-specific widgets here
            ],
          );
        } else {
          return Row(
            children: [
              Text('Large Screen'),
              // Add desktop-specific widgets here
            ],
          );
        }
      },
    );
  }
}
```

Routing And Navigation In Web Applications

Web applications require effective routing and navigation. Flutter provides a robust navigation system that works well for web applications.

- **Named Routes**: Define named routes for easy navigation between different pages.
- **Router**: Use the Router widget to manage complex navigation scenarios.

Example of Named Routes:

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

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

class MyApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Flutter Web App',
     initialRoute: '/',
     routes: {
       '/': (context) => HomeScreen(),
       '/about': (context) => AboutScreen(),
     },
   );
 }
}

class HomeScreen extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(title: Text('Home')),
     body: Center(
       child: ElevatedButton(
         onPressed: () {
           Navigator.pushNamed(context, '/about');
         },
         child: Text('Go to About'),
       ),
     ),
   );
 }
}

class AboutScreen extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(title: Text('About')),
     body: Center(child: Text('This is the About page')),
   );
 }
}
```

15.5 BUILDING DESKTOP APPLICATIONS WITH FLUTTER

Setting Up the Desktop Environment

To build Flutter applications for desktop, ensure you have the following:

1. **Flutter SDK**: Ensure that you have the latest Flutter SDK installed.
2. **Desktop Support**: Enable desktop support in Flutter by running the following command:

   ```
   flutter config --enable-windows-desktop
   flutter config --enable-macos-desktop
   flutter config --enable-linux-desktop
   ```

Creating a Flutter Desktop Project

Creating a desktop project is similar to creating a web project. Use the following command:

```
flutter create my_desktop_app
```

Navigate to the project directory:

```
cd my_desktop_app
```

To run the desktop application, execute:

```
flutter run -d windows   # For Windows
flutter run -d macos     # For macOS
flutter run -d linux      # For Linux
```

Desktop-Specific Features

Flutter for desktop supports several features that enhance the user experience:

- **Window Management**: Control the window size, position, and behavior of the application.
- **Mouse and Keyboard Events**: Respond to mouse and keyboard events to enhance interactivity.
- **Native APIs**: Access native features through platform channels to use platform-specific functionality.

Creating a Desktop-Responsive UI

Building a responsive UI is essential for desktop applications as well. Use the following techniques:

- **Flexible Widgets**: Use Flexible and Expanded widgets to create layouts that adapt to the window size.
- **Custom Scrollable Areas**: Implement scrollable areas for content that exceeds the window size.

Example of a Responsive Desktop Layout:

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

class DesktopResponsiveLayout extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Desktop App')),
      body: Row(
        children: [
          NavigationDrawer(), // Custom navigation drawer
          Expanded(child: ContentArea()), // Main content area
        ],
      ),
    );
  }
}
```

15.6 BEST PRACTICES FOR FLUTTER WEB AND DESKTOP DEVELOPMENT

Performance Optimization

- **Lazy Loading**: Implement lazy loading for images and data to improve initial load times.
- **Minimize HTTP Requests**: Reduce the number of HTTP requests by caching data and optimizing API calls.

Accessibility

Ensure that your application is accessible to all users by:

- Using semantic widgets (e.g., Semantics) to provide additional context.
- Testing with screen readers to ensure the app is usable for visually impaired users.

Testing and Debugging

- **Use the Flutter DevTools**: Leverage Flutter DevTools for debugging and performance profiling.
- **Responsive Testing**: Test your application on various screen sizes and resolutions to ensure proper responsiveness.

15.7 DEPLOYMENT AND PUBLISHING

Web Deployment

To deploy a Flutter web application:

1. Build the web app:
   ```
   flutter build web
   ```
2. Deploy the contents of the build/web directory to a web server (like Firebase Hosting, GitHub Pages, or your own server).

Desktop Application Distribution

To distribute desktop applications:

- **Windows**: Create an installer using tools like Inno Setup or NSIS.
- **macOS**: Use pkg or dmg files to package the application.
- **Linux**: Create .deb or .rpm packages for distribution.

Check Your Progress

1. What is the primary advantage of using Flutter for web and desktop development?

A) Requires different codebases for each platform

B) Offers a single codebase for multiple platforms

C) Lacks a rich set of widgets

D) Provides limited performance optimization

2. Which command is used to run a Flutter web application in the Chrome browser?

A) flutter build web

B) flutter run -d web

C) flutter run -d chrome

D) flutter start web

3. Which Flutter widget is used to create a responsive layout that adapts to different screen sizes?

A) Container

B) Column

C) LayoutBuilder

D) Center

4. What is the purpose of using MediaQuery in a Flutter web application?

A) To manage the navigation between different screens

B) To access device information and layout dimensions

C) To handle user input from forms

D) To define global styles for the application

5. When deploying a Flutter web application, which command should you use to build the production-ready version?

A) flutter package build web

B) flutter run web

C) flutter build web

D) flutter deploy web

15.8 LET US SUM UP

Flutter for web and desktop expands the possibilities for developers, allowing them to create high-quality, cross-platform applications from a single codebase. With its rich set of widgets, responsive design capabilities, and support for various platforms, Flutter is an excellent choice for modern application development. By following best practices and leveraging the tools provided by the Flutter ecosystem, you can build engaging and performant applications that reach a wider audience across different platforms.

As the Flutter ecosystem continues to evolve, staying up-to-date with the latest features and enhancements will further empower you to create remarkable applications that provide a seamless user experience, whether on the web or desktop.

15.9 CHECK YOUR PROGRESS: POSSIBLE ANSWERS

1- B) Offers a single codebase for multiple platforms

2- C) flutter run -d chrome

3- C) LayoutBuilder

4- B) To access device information and layout dimensions

5- C) flutter build web

15.10 FURTHER READING

- https://docs.flutter.dev/platform-integration/web/building
- https://flutter.dev/multi-platform/web
- https://flutter.dev/multi-platform/desktop
- https://docs.flutter.dev/platform-integration/desktop
- https://blog.flutter.wtf/desktop-app-with-flutter/

15.11 ASSIGNMENT

1. Discuss the differences in the user interface design considerations between
2. Flutter mobile applications and Flutter web applications. What factors must be taken into account when designing for different platforms?
3. Explain the role of the Navigator widget in Flutter web applications. How does it facilitate navigation, and what are the key differences in navigation patterns between web and mobile applications?

4. Describe how you would implement a responsive layout for a Flutter web application. What tools and widgets would you use, and how would you ensure that the layout adapts effectively to various screen sizes?
5. What strategies would you employ to optimize the performance of a Flutter desktop application? Discuss specific techniques that can be applied to improve load times and responsiveness.
6. Imagine you are tasked with deploying a Flutter web application to a production environment. Outline the steps you would take to prepare for deployment, including any tools or services you would consider using for hosting.

15.12 ACTIVITIES

- Create a basic web application using Flutter that demonstrates understanding of web development concepts.
- Learn and apply principles of responsive design in Flutter.
- Develop a simple desktop application using Flutter.
- Practice making API calls in a Flutter web application.
- Conduct user testing to gather feedback on a Flutter web or desktop application.
- Learn the steps required to deploy a Flutter web application.