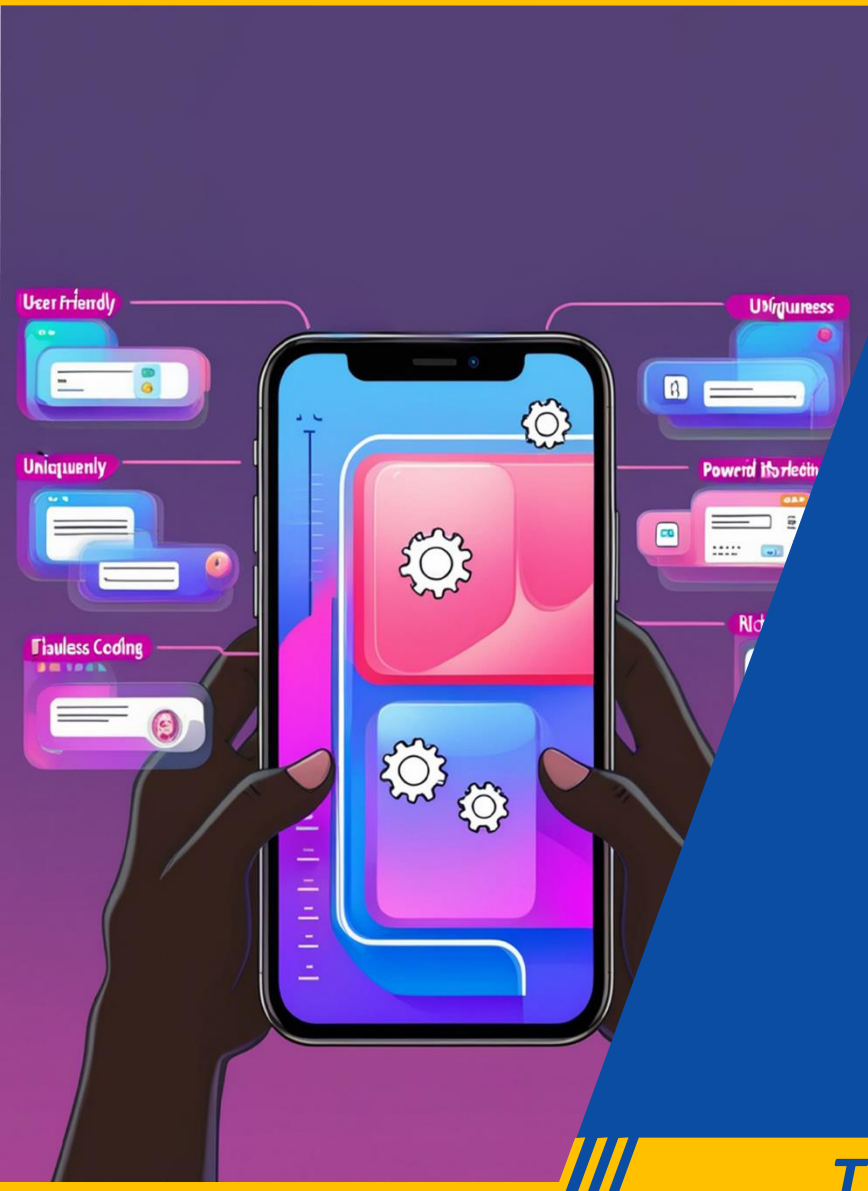
RQFL EVEL 5



SWDMA501

SOFTWARE DEVELOPMENT

Mobile Application Development

TRAINEE'S MANUAL

October, 2024

MOBILE APPLICATION DEVELOPMENT

KOICA Korea International Cooperation Agency

TQUM TVET Quality Management Project

2024

AUTHOR'S NOTE PAGE (COPYRIGHT)

Copies available from:

- *HQs: Rwanda TVET Board-RTB*
- *Web: www.rtb.gov.rw*
- **KIGALI-RWANDA**

Original published version: October, 2024

ACKNOWLEDGEMENTS

The publisher would like to thank the following for their assistance in the elaboration of this training manual:

Rwanda TVET Board (RTB) extends its appreciation to all parties who contributed to the development of the trainer's and trainee's manuals for the TVET Certificate V in Software Development, specifically for the module "**SWDMA501: Mobile Application Development** ".

We extend our gratitude to KOICA Rwanda for its contribution to the development of these training manuals and for its ongoing support of the TVET system in Rwanda.

We extend our gratitude to the TQUM Project for its financial and technical support in the development of these training manuals.

We would also like to acknowledge the valuable contributions of all TVET trainers and industry practitioners in the development of this training manual.

The management of Rwanda TVET Board extends its appreciation to both its staff and the staff of the TQUM Project for their efforts in coordinating these activities.

This training manual was developed:

Under Rwanda TVET Board (RTB) Guiding Policies and Directives

Under Financial and Technical Support of

COORDINATION TEAM

RWAMASIRABO Aimable

MARIA Bernadette M. Ramos

MUTIJIMANA Asher Emmanuel

Production Team

Authoring and Review

HABANABAKIZE Jerome

TUYISINGIZE Leopord

NDAGIJIMANA Enock

Validation

MUKASHYAKA Christine

MUZIMANGANYE Jean Pierre

Conception, Adaptation and Editorial works

HATEGEKIMANA Olivier

GANZA Jean Francois Regis

HARELIMANA Wilson

NZABIRINDA Aimable

DUKUZIMANA Therese

NIYONKURU Sylvestre

NTEZIMANA Aimable

Formatting, Graphics, Illustrations, and infographics

YEONWOO Choe

SUA Lim

SAEM Lee

SOYEON Kim

WONYEONG Jeong

SHYAKA Emmanuel

Financial and Technical support

KOICA through TQUM Project

TABLE OF CONTENT

AUTHOR'S NOTE PAGE (COPYRIGHT) ---------- iii

ACKNOWLEDGEMENTS ---------- iv

TABLE OF CONTENT ---------- vii

ACRONYMS ---------- ix

INTRODUCTION ---------- 1

MODULE CODE AND TITLE: SWDMA501 MOBILE APPLICATION DEVELOPMENT ---------- 2

Learning Outcome 1: Apply Basics of Dart ---------- 3

- Key Competencies for Learning Outcome 1: Apply Basics of Dart ---------- 4
- Indicative content 1.1: Preparation of Development Environment ---------- 6
- Indicative content 1.2: Applying the Dart Concept ---------- 19
- Indicative content 1.3: Applying Object Oriented Programming (OOP) ---------- 31
- Indicative content 1.4: Using Dart Libraries and Packages ---------- 41
- Learning outcome 1 end assessment ---------- 45
- References ---------- 47

Learning Outcome 2: Implement UI Designs ---------- 48

- Key Competencies for Learning Outcome 2: Implement UI designs ---------- 49
- Indicative content 2.1: Preparation of flutter environment ---------- 51
- Indicative content 2.2: Applying flutter's widget system ---------- 70
- Indicative content 2.3: Implementation of State Management ---------- 81
- Indicative content 2.4: Using Pre-designed Widgets ---------- 95
- Learning outcome 2 end assessment ---------- 107
- References ---------- 109

Learning Outcome 3: Integrate Backend Functionality ---------- 110

- Key Competencies for Learning Outcome 3: Integrate Backend Functionality. ---------- 111
- Indicative content 3.1:Integration of external services ---------- 113
- Indicative content 3.2 Implement storage management ---------- 140
- Indicative content 3.3: Implementation of Microapps ---------- 148
- Indicative content 3.4: Perform Error Handling ---------- 155
- Indicative content 3.5: Perform Testing ---------- 162
- Indicative content 3.6 Debug Codebase issues ---------- 172
- Learning outcome 3 end assessment ---------- 179
- References ---------- 182

Learning Outcome 4: Publish Application----------183
Key Competencies for Learning Outcome 4: Publish Application ---------- 184
Indicative content 4.1: Generation of installable files ---------- 186
Indicative content 4.2: Submission of application files ---------- 197
Indicative content 4.3: Address Post eployment issues ---------- 216
Learning outcome 4 end assessment ---------- 223
References ---------- 226

ACRONYMS

AAB: Android App Bundle

API: Application Programming Interface

APK: Android Package Kit

CLI: Command Line Interface

E2E: End-to-End

IDE: Integrated Development Environment

iOS: iPhone Operating System

IoT: Internet of Things

IPA: iOS package App Store

KOICA: Korean International Cooperation Agency

OOP: Object Oriented Programming

ORM: Object-Relational Mapping

OS: Operating System

PWA: Progressive Web Apps

RTB: Rwanda TVET Board

SDK: Software Development Kit

SPA: Single Page Applications

TQUM Project: TVET Quality Management Project

INTRODUCTION

This trainee's manual includes all the knowledge and skills required in software development specifically for the module of **"Mobile Application Development".** Trainees enrolled in this module will engage in practical activities designed to develop and enhance their competencies. The development of this training manual followed the Competency-Based Training and Assessment (CBT/A) approach, offering ample practical opportunities that mirror real-life situations.

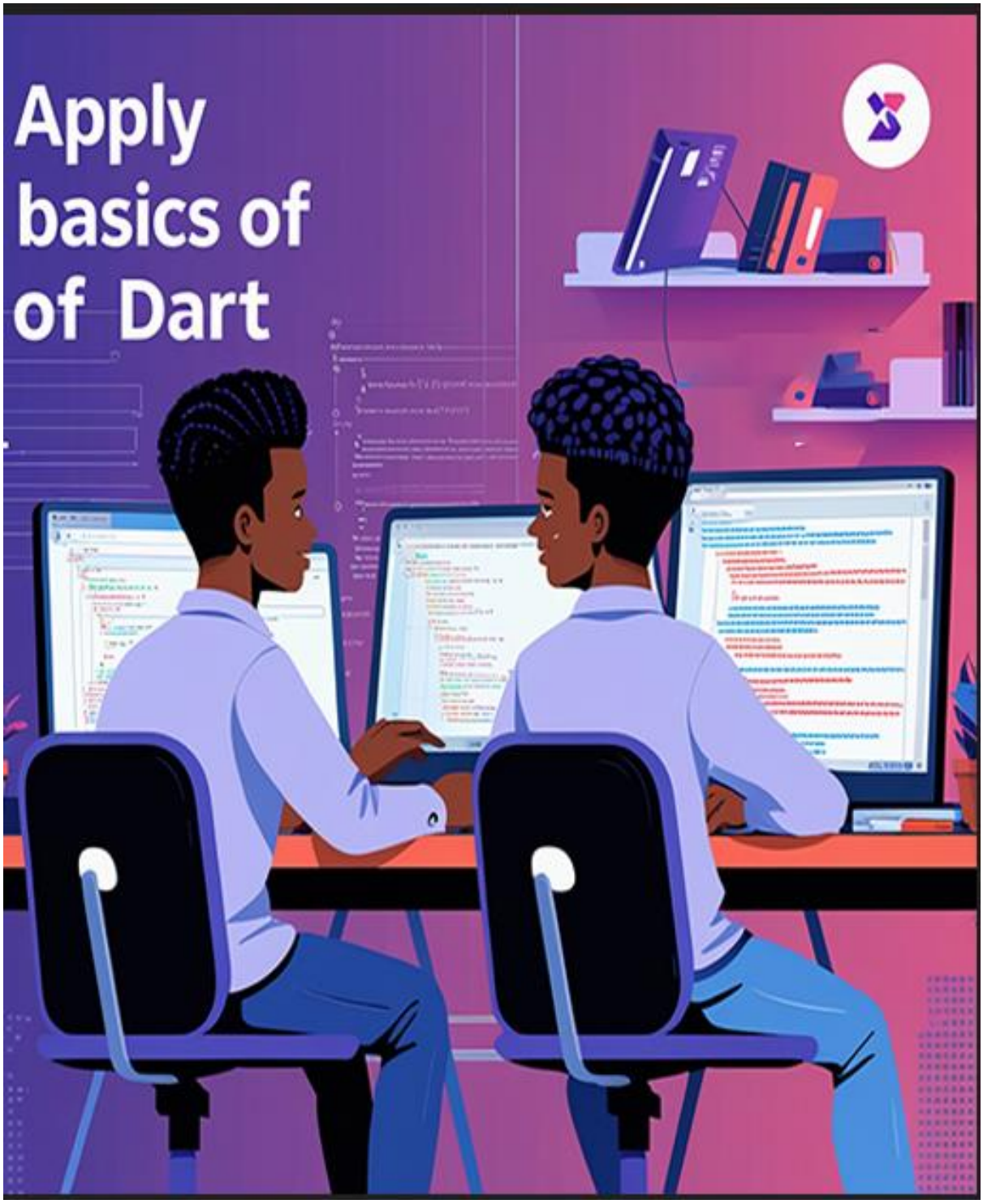
The trainee's manual is organized into Learning Outcomes, which is broken down into indicative content that includes both theoretical and practical activities. It provides detailed information on the key competencies required for each learning outcome, along with the objectives to be achieved.

As a trainee, you will start by addressing questions related to the activities, which are designed to foster critical thinking and guide you towards practical applications in the labour market. The manual also provides essential information, including learning hours, required materials, and key tasks to complete throughout the learning process.

All activities included in this training manual are designed to facilitate both individual and group work. After completing the activities, you will conduct a formative assessment, referred to as the end learning outcome assessment. Ensure that you thoroughly review the key readings and the 'Points to Remember' section.

MODULE CODE AND TITLE: SWDMA501 MOBILE APPLICATION DEVELOPMENT

Learning Outcome 1: Apply Basics of Dart
Learning Outcome 2: Implement UI Designs
Learning Outcome 3: Integrate Backend Functionality
Learning Outcome 4: Publish Application


Apply
basics of
of Dart

Indicative Contents
1.1 Preparation of Development Environment
1.2 Applying the Dart Concept
1.3 Applying Object Oriented Programming (OOP)
1.4 Using Dart Libraries and Packages

Key Competencies for Learning Outcome 1: Apply Basics of Dart

Knowledge	Skills	Attitudes
● Description of development environment. ● Description of dart concept. ● Description of Object-Oriented Programming (OOP) in Dart ● Description of dart libraries and packages	● Installing of key tools (Windows and Apple) ● Applying the dart concept ● Applying Object-oriented programming (OOP) in Dart. ● Using dart libraries and packages	● Having Teamwork ability ● Having Passion for Learning ● Having good Collaboration and Communication ● Being attentive for using dart libraries and packages.

Duration: 20 hrs

Learning outcome 1 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe correctly the development of environment as used in dart language
2. Describe effectively the dart concept used in dart language.
3. Apply correctly the dart concept as applied in mobile application.
4. Install properly the key tools as applied in Mobile application.
5. Describe correctly the Object-Oriented Programming (OOP) used in dart language.
6. Apply correctly the Object-oriented programming (OOP) in Dart language.
7. Use properly dart libraries and packages as applied in mobile application development.

 **Resources**

Equipment	Tools	Materials
● Computer	● VS Code ● Android Studio ● Dart SDK	● Third party libraries ● Internet ● Electricity

Indicative content 1.1: Preparation of Development Environment

Duration: 5 hrs

Theoretical Activity 1.1.1: Description of Dart concepts

Tasks:

1. You are requested to answer the following questions:
   i. Describe the following key terms:
      - ✓ Dart language
      - ✓ Data type
      - ✓ Variable
      - ✓ Native apps
      - ✓ Cross platform
   ii. Describe dart features and characteristics
   iii. Identify use cases of dart language.
   iv. Differentiate control flow structure and function
2. Write your findings to flipchart/papers
3. Present the findings to the whole class or trainer
4. For more clarification, read the key readings 1.1.1. In addition, ask questions w here necessary.

Key readings 1.1.1: Description of dart concepts

✓ **Introduction to Dart**

Dart is a modern, object-oriented programming language developed by Google. It is specifically designed for building high-performance, cross platform applications. Dart offers a clean syntax, strong typing, and a wide range of built-in libraries, making it a powerful tool for app development.

Dart features and characteristics

✓ **Flutter Framework: Dart** is the primary language used for developing mobile applications with Flutter. Flutter is a UI toolkit developed by Google that allows you to create natively compiled applications for mobile, web, and desktop from a single codebase. Flutter has gained popularity for its hot reload feature, expressive UI, and excellent performance.

✓ **Cross-Platform Development:** Dart, in conjunction with Flutter, enables you to build cross-platform applications.

This means you can write code once and run it on both iOS and Android platforms, saving time and effort compared to developing separate codebases for each platform.

- ✓ **Fast Development:** Dart's hot reload feature in Flutter allows developers to instantly see the effect of the changes made in the code without restarting the entire application. This significantly speeds up the development process and makes it easier to experiment with different UI designs and features.
- ✓ **Strongly Typed Language:** Dart is a statically typed language, which means that it helps catch errors during development before the code is executed. This can lead to more reliable and maintainable code, especially in larger projects.
- ✓ **Asynchronous Programming**: Dart has built-in support for asynchronous programming, making it well-suited for tasks that involve handling multiple operations simultaneously. This is particularly useful for developing responsive and efficient applications, especially in scenarios like web development.
- ✓ **Growing Ecosystem:** Dart's ecosystem is growing, with an increasing number of packages and libraries available through the Dart Package Manager (pub.dev). This can save developers time by leveraging existing solutions for common tasks.
- ✓ **Google's Support:** Dart is backed by Google, and the company actively uses it in various projects, including some high-profile ones. This support provides a level of confidence in the language's stability and future development.
- ✓ **Server -Side Development:** While Dart is widely known for client-side development with Flutter, it can also be used for server-side development. The Dart SDK includes libraries for building server-side applications, making it a versatile language for full-stack development.

Dart Framework

Dart has several frameworks that cater to different aspects of development, especially for building web, mobile, and server-side applications.

Here's an overview of popular Dart frameworks:

1. Flutter
2. Aqueduct
3. Shelf
4. Angel
5. Jaguar
6. GetX

Description of listed dart frameworks

✓ **Flutter**

Purpose: Cross-platform mobile, web, and desktop applications.

Features:

- Rich set of pre-designed widgets for building modern UIs.
- Hot-reload for quick code changes.

- Strong community and ecosystem.
- Allows building apps for Android, iOS, web, macOS, Windows, and Linux.
- Good performance due to Dart's ahead-of-time (AOT) compilation.
- Use Case: Ideal for building visually attractive and highly performant mobile apps.
  Examples: Google Pay, Alibaba, and Reflect.

✓ **Aqueduct (deprecated)**

Purpose: Web framework for building RESTful APIs.

Features:

- Built-in ORM for interacting with databases.
- Dependency injection and middleware support.
- Testing utilities for API development.
- Use Case: Used for backend web services and REST API development in Dart, but currently deprecated. For similar purposes, developers might opt for more general-purpose solutions like `shelf`.

✓ **Shelf**

Purpose: Simple web server framework.

Features:

- Lightweight and flexible.
- Provides a minimalistic structure for handling HTTP requests.
- Middleware support for request/response manipulation.
- Use Case: Ideal for creating small web servers, microservices, and API servers with custom routing and middleware.

✓ **Angel**

Purpose: Full-stack server framework.

Features:

- Includes an ORM, templating engine, and real-time capabilities.
- Supports GraphQL, WebSockets, and REST.
- Modular architecture for easy plugin management.
- Use Case: Useful for building both server-side applications and APIs with built-in support for multiple protocols like HTTP, GraphQL, etc.

✓ **Jaguar**

Purpose: Dart framework for building APIs.

Features:

- Route handling, middleware support, and ORM for database interaction.
- Built for speed and performance.
- Strongly focused on RESTful API design.

- Use Case: Fast web framework designed for building APIs and backend services.

✓ **GetX (for Flutter)**

Purpose: State management, navigation, and dependency injection for Flutter.

Features:

- Powerful state management and easy navigation system.
- Minimal boilerplate code.
- Lightweight, and highly performant.
- Use Case: Commonly used in Flutter projects for managing state, routes, and dependencies.

Note that: These frameworks allow Dart to cover both front-end and back-end development, making it versatile for a wide range of applications.

Use cases of dart language

✓ **Cross-Platform Mobile App Development (Flutter)**

Use Case: Dart is the primary language for Flutter, Google's popular framework for building cross-platform mobile applications.

Example: Building apps for both Android and iOS with a single codebase, such as Google Pay or Reflectly.

✓ **Web Development**

Use Case: Dart can be compiled to JavaScript, making it suitable for building client-side web applications.

Example: Progressive web apps (PWAs) and Single Page Applications (SPAs) using Dart's Flutter for Web.

✓ **Desktop Applications (Flutter Desktop)**

Use Case: With Flutter, Dart can also be used to develop desktop applications for Windows, macOS, and Linux.

Example: Creating desktop versions of mobile apps using the same Dart codebase.

✓ **Backend Development (Web Servers and APIs)**

Use Case: Dart has server-side frameworks like Shelf, Angel, and Jaguar that can be used to build web servers and RESTful APIs.

Example: Developing microservices or APIs for mobile applications.

✓ **Game Development**

Use Case: Dart can be used in game development, often in conjunction with Flutter for casual games or simple 2D games.

Example: Creating mobile games with Flame, a 2D game engine for Flutter.

✓ **IoT Development**

Use Case: Dart's scalability and cross-platform support make it suitable for Internet of Things (IoT) projects where mobile or web apps interact with hardware.

Example: Building control apps for smart home devices.

- ✓ **Command-Line Tools**

  Use Case: Dart can be used to create command-line tools or scripts for automation tasks, with Dart's strong package ecosystem aiding in CLI development.

  Example: Writing CLI tools for task automation, file processing, or package management.
- ✓ **Full-Stack Development**

  Use Case: Dart can be used for full-stack development, handling both front-end (with Flutter for Web or Mobile) and back-end (with Shelf or Angel).

  Example: Creating full web and mobile applications with a Dart-powered backend and frontend.
- ✓ **Real-Time Applications**

  Use Case: Dart can be used to build real-time applications such as chat applications, especially when using WebSockets.

  Example: Developing real-time messaging systems or collaborative apps using Dart and Flutter.
- ✓ **Embedded Systems**

  Use Case: While not as common as other languages in this domain, Dart's performance and small footprint can be utilized in embedded systems when combined with Flutter.

  Example: Developing companion apps that interact with embedded hardware.

2. Description of key terms

- **Data type**

Data type refer to the classification or categorization of data items based on the kind of value they hold and the operations that can be performed on them. In programming, data types specify what type of data a variable can store, such as numbers, text, or more complex structures like lists or maps. They help the compiler or interpreter understand how to handle and process the data correctly.

Example: int, double, String, Boolean, Lists, Map, Set

- **Variables**

In Dart, **variables** are containers that store data values which can be referenced and manipulated throughout a program. A variable has a name, a type (explicit or inferred), and can hold different types of data such as numbers, strings, or objects. Dart is statically typed, meaning every variable must have a type, but the type can be inferred or explicitly defined.

- **Control flow Structures**

Control flow structures are used to determine the flow of execution in a program based on conditions or repetitive tasks. These include conditional statements, loops, and branching.

- **Functions**

A **function** in Dart is a block of code that performs a specific task. Functions help to make code modular, reusable, and organized. Dart supports both named and anonymous functions.

- **Native Apps**

Native Apps refer to applications developed specifically for a particular operating system or platform using the native programming languages and tools provided by that platform. These apps are designed to run directly on the operating system and take full advantage of its features and functionalities.

- **Cross Platform**

Cross-Platform refers to the development of applications that can run on multiple operating systems or platforms using a single codebase. This approach allows developers to write the app once and deploy it across different platforms, such as iOS and Android.

Practical Activity 1.1.2: Installing key tools (Windows and Apple)

Task:

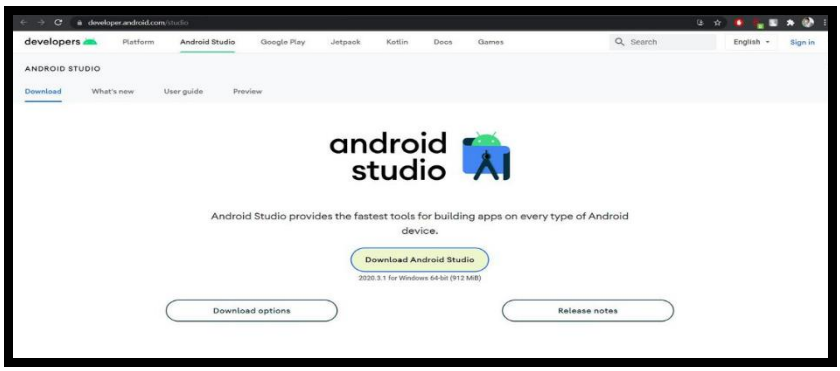
1: Read Key reading 1.1.2.
2: Referring to the key reading 1.1.2, You are requested to go to the computer lab to install key tools (android studio and vs code).
3: Present your work to the trainer/ your colleagues.
4: Ask for clarification if any.

Key readings 1.1.2: Installing key tools (Visual studio code, Android studio and Dart SDK)

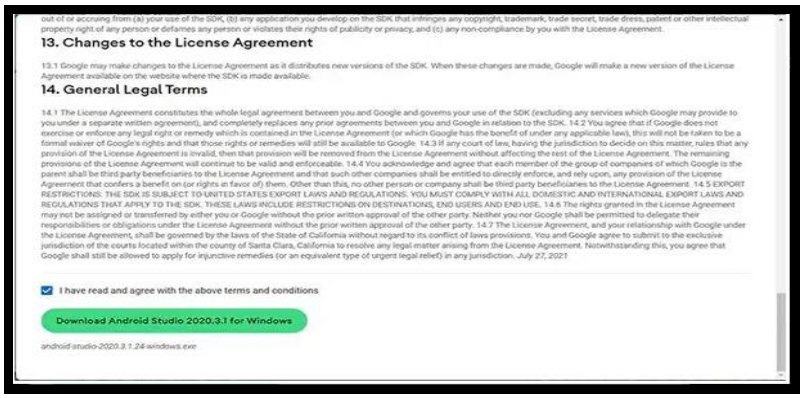
✓ **How to Install Android studio on Windows?**

Step 1: Setting up Android tools and emulator for android devices.

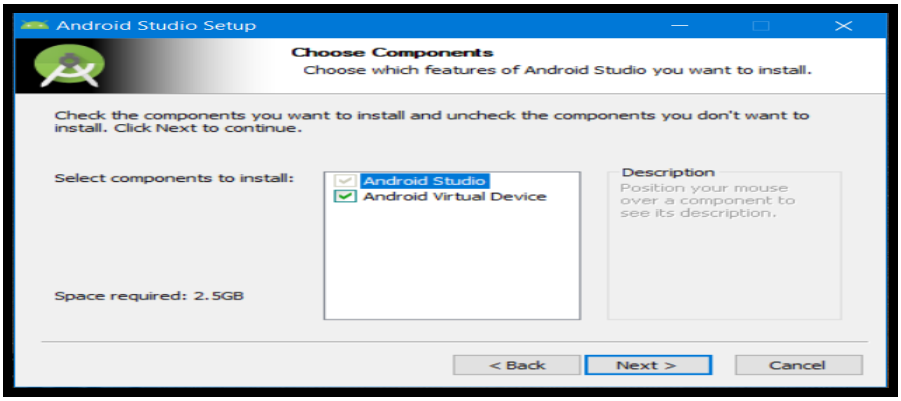
The first step is to download and install Android Studio. To do this, navigate to the official page of **Android Studio** and click on '**Download Android Studio**'.



After accepting the license agreements, you are good to go! Click on the final **Download** button to start downloading.

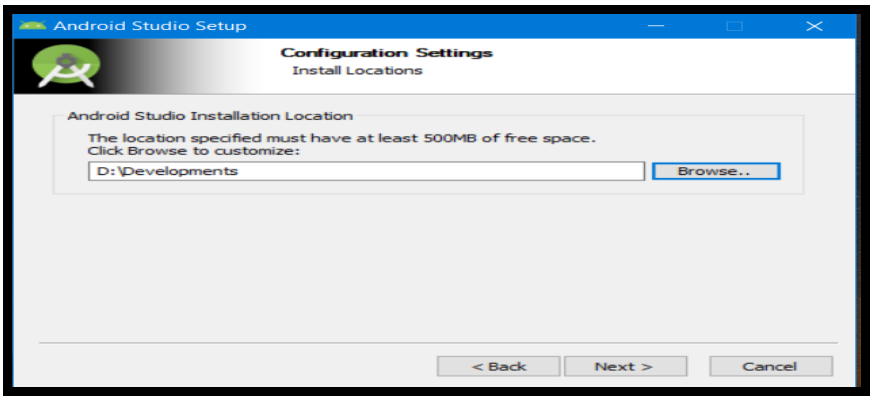


After the download is complete, let's move on to the next step, i.e. installation. Under '**Components** ', make sure that both Android Studio and Android Virtual Device are checked, and only then proceed. The Android Virtual Device is an essential tool for running various types and sizes of android emulators to test your flutter project. Henceforth, click on '**Next**'.



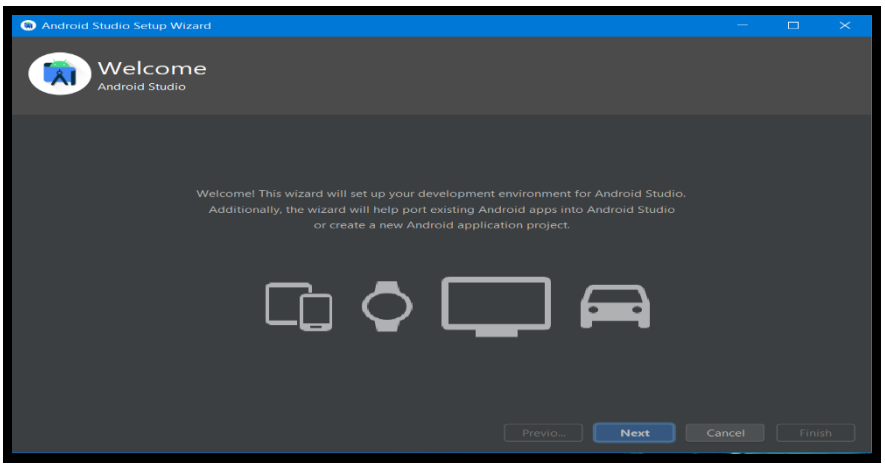
Select the directory you would want your file to be installed in. It is recommended to select some other path apart from the system drive. Once done, click on '**Next**'.

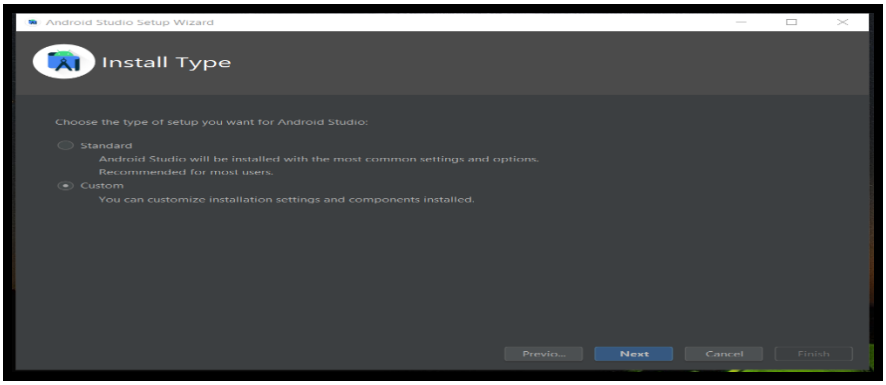
Step 2: Install android studio



Finally, click on '**Install**'. Wait for a couple of seconds for the installation to complete. Check the box beside '**Launch Android Studio**'. Click on '**Finish**'.

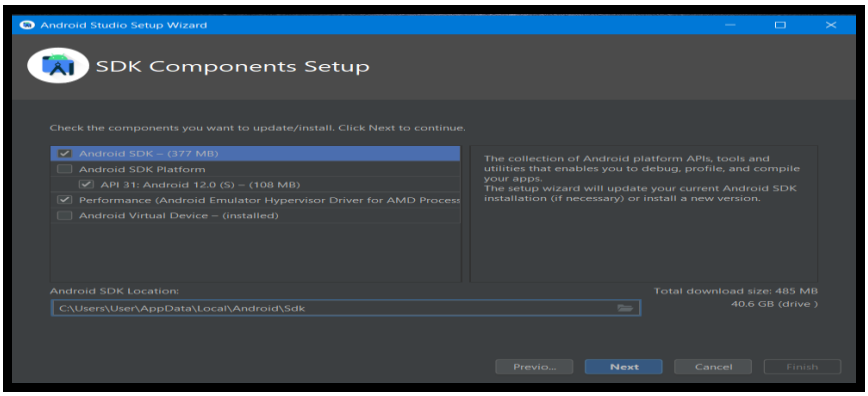
Wait for Android Studio to launch on your computer. On the home screen, click **Next > Custom > Next**.



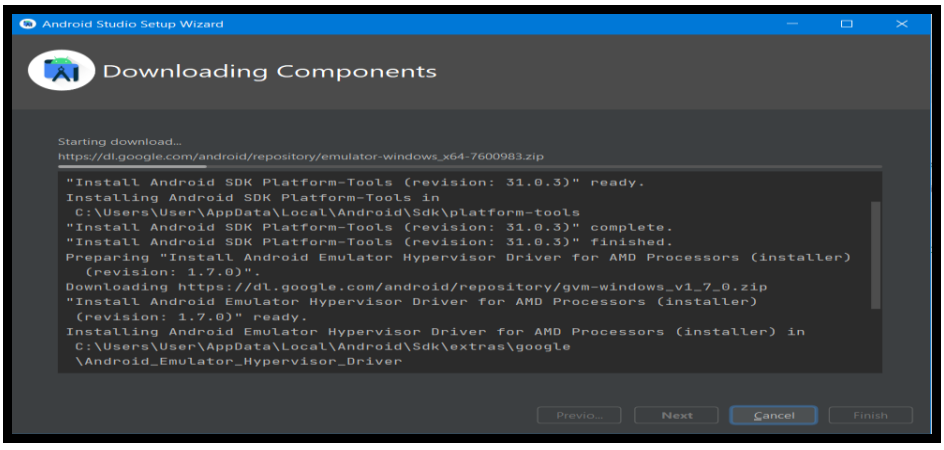


For the Java Development kit location in the next step, it is recommended to keep the default path it requires, to avoid the hassle. In the next step, choose the UI appearance you'd like for Android Studio. Click '**Next**'.

This next step is a bit important. Remember to check the required boxes exactly as shown below. If kits have already been installed, you can ignore those and move on. Click '**Next**'. Set your desired folder for Android SDK.



With that done, click on '**Finish**'. Android Studio will now install all the necessary android tools required for the execution of your flutter projects. This may take a significant time – it's better to wait!



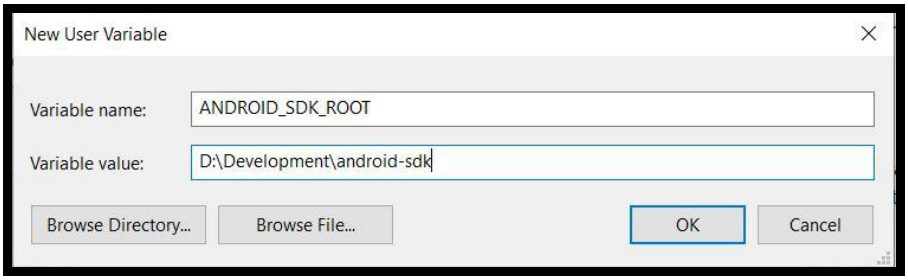
Now, we are ready to create and build flutter projects on Android Studio and run it on a real or a virtual Android device (emulator).

Step 3: Set SDK as an environment variable, for global access.

Now, open Command Prompt terminal and run 'flutter doctor' again. If you have

installed Android SDK in the default directory suggested by Android Studio, there wouldn't be any problem that would appear. Nevertheless, if you have installed it in a non-default directory, flutter would not be able to detect it in your system. To help it able to do that, you guessed it…we would be assigning it as an environment variable, **giving global access**.

As discussed earlier in Step 4, go to environment variables and click '**New**', and do the following (as recommended by flutter doctor). Click '**OK**'.



Step 4: Accept required Android Licenses.

On the Command Prompt terminal, type in:

flutter doctor --android-licenses

as suggested by flutter doctor. Hit Enter. To review licenses, type '**y**' for Yes.

You'll see a couple of repeated prompts that look like this:

Accept? (y/N):

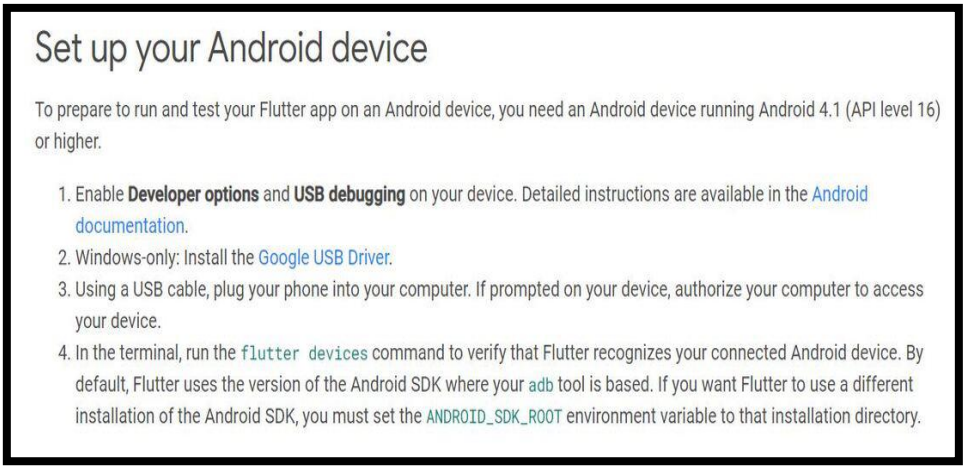
Type '**y**' whenever asked for.

Finally, after all the license agreements have been accepted, you should see a message that looks something like this:

All SDK package licenses accepted

Step 5: Setup Android Emulator.

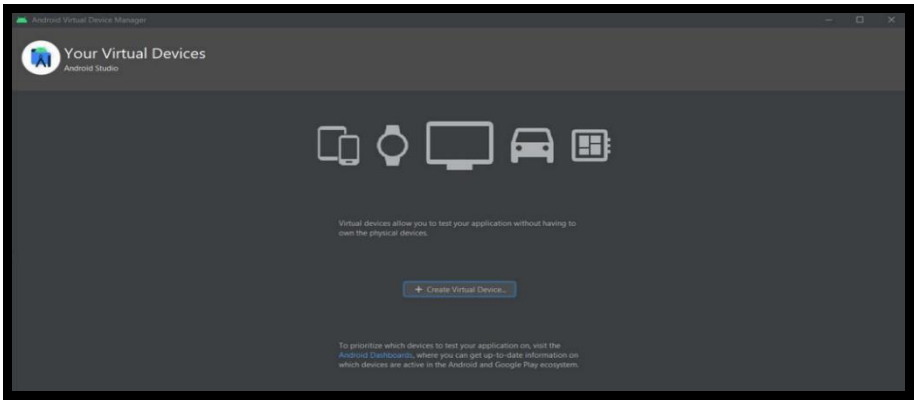
You have the option to choose between an Android Device or an Android Emulator to build your application on. It depends totally on you.



For setting up Android Emulator, you need to go through the following steps:

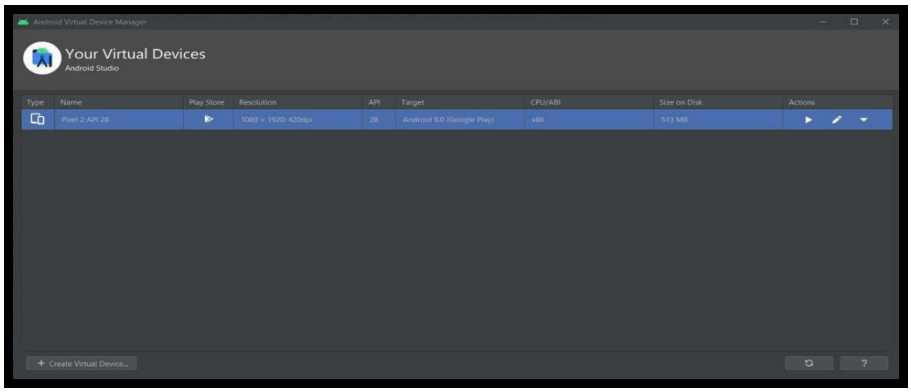
- Open **Android Studio.**
- On the topmost menu bar, click on **Tools > SDK Manager.**

- Verify whether you have the latest SDK installed. Remember to install the latest stable version too by checking on the box to the left. In my case, it is '**Android 9.0 (Pie)**'. *You can even uncheck the latest version (if not stable), to not only save space but also run all your applications on the stable version itself.*
- Under the '**SDK Tools**' tab, don't forget to check Google USB Driver to later connect a real Android Device. With that, click '**Apply**'. Click '**OK**' to start SDK installation.
  This might take a couple of minutes to complete. After the setup is done, click on '**Finish**'. Your setup is now complete!
- To have a first look at your Android Emulator, open Android Studio. Go to **Tools > AVD Manager**. A dialog box appears.



- Click on '**Create Virtual Device…**', select a device and its dimensions according to your preference, select a system image and lastly, under all default settings,

  click on 'Finish'. Click on the ' ' button to fire up your emulator.



There you go! You now have a fully functional flutter framework with devices/emulators to build your beautiful apps on.

Installation of virtual studio code on windows

Step1. Download: Get the installer from https://code.visualstudio.com.

Step2. Accept License: Agree to the terms.

Step3. Choose Location: Pick an installation folder (default is fine).

Step4. Select Options: (Recommended)

- ✓ Add "Open with Code" to context menu
- ✓ Add to PATH for terminal access

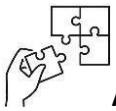
Step5. Install: Click Install and wait for it to finish.

Step6. Launch & Verify: Open VS Code and confirm by typing code in the command prompt.

Points to Remember

- **Dart** is a modern, object-oriented programming language developed by Google.
- Dart Features and Characteristics
  - Flutter Framework
  - Cross-Platform Development
  - Object-Oriented Programming
- There are different Use Cases of Dart:
  - ✓ Mobile Apps
  - ✓ Web Apps
  - ✓ Desktop Apps
- Dart Frameworks:
  - ✓ **Flutter:** for building mobile, web, and desktop apps with Dart.
  - ✓ **Angel:** Full-stack server-side framework supporting HTTP, GraphQL, and WebSockets.
  - ✓ **Shelf:** Lightweight web server for creating HTTP APIs and microservices.
- Description of the following key terms:
  - ✓ **Data Type:** Defines the type of data a variable can hold.
  - ✓ **Variables:** Containers for storing data values.
  - ✓ **Control Flow Structures:** Used for decision-making and looping .
  - ✓ **Functions:** Blocks of code that perform specific tasks.
  - ✓ **Native Apps:** Applications specifically built for a single platform (e.g., iOS or Android) using native tools.
  - ✓ **Cross-Platform:** Apps that run on multiple platforms from a single codebase (e.g., Flutter apps for both iOS and Android).
- While installing Android studio pass through the following steps:
  - ✓ Download android studio setup.
  - ✓ Install android studio setup.
  - ✓ Set SDK
  - ✓ Accept required android licenses.
  - ✓ Set up Android emulator.

- **While installing Visual Studio code pass through the following steps:**
  - ✓ Download Visual studio code setup.
  - ✓ Accept license.
  - ✓ Choose location.
  - ✓ Select option.
  - ✓ Install Visual studio code setup.
  - ✓ Launch and verify.



Application of learning 1.1.

The **Free Company** assigned you to create a cross-platform mobile application using Flutter and Dart. To ensure the application works properly on Windows or macOS, you need to set up a development environment on one of those operating systems.
As mobile developer you are requested to help the company to install the key tools (Visual studio code and Android studio).

Indicative content 1.2: Applying the Dart Concept

Duration: 5 hrs

Practical activity 1.2.1: Performing variable declaration

Tasks:

1: Read Key reading 1.2.1

2: With referring to the key reading 1.2.1, You are requested to go to the computer lab to p erform a program using variable declaration in Dart programming language.

3: Present your work to the trainer/ your colleagues.

4: Ask for clarification if any.

Key readings 1.2.1.: Performing variable declaration

Here are the steps to implement variable declaration, data types, and naming conventions in Dart:

Step 1: Variable Declaration

✓ **Using var:**

- Use var when you want the Dart analyser to infer the type based on the value assigned.
- Example:

```
var name = 'Alice'; // Inferred as String
var age = 30;       // Inferred as int
```

✓ **Using final:**

- Use final for variables that should only be set once.
- Example:

```
final birthYear = 1993; // Cannot be changed later
```

✓ **Using const:**

- Use const for compile-time constants that cannot change.
- Example:

```
const pi = 3.14; // Compile-time constant
```

Step 2: Data Types

✓ **Numbers:**

- Declare integers and doubles.
- Example:

```
int age = 30;          // Integer
double price = 19.99;  // Double
```

✓ **Strings:**

- Declare strings using single, double, or triple quotes.
- Example:

```
String greeting = 'Hello, World!'; // Single-quoted string
String multiLine = '''This is a
multi-line string.''';                // Multi-line string
```

✓ **Booleans:**

- Declare boolean values.
- Example:

```
bool isActive = true; // Boolean
```

✓ **Lists:**

- Declare lists of various types.
- Example:

```
List<String> fruits = ['Apple', 'Banana', 'Cherry']; // List of Strings
List<int> numbers = [1, 2, 3, 4, 5]; // List of integers
```

✓ **Maps:**

- Declare maps with key-value pairs.
- Example:

```
Map<String, int> scores = {'Alice': 90, 'Bob': 85}; // Map with String keys and int valu
es
```

✓ **Sets:**

- Declare sets of unique items.
- Example:

```
Set<String> uniqueNames = {'Alice', 'Bob', 'Charlie'}; // Set of Strings
```

✓ **Dynamic:**

- Use dynamic for variables that can change type.
- Example:

```
dynamic variable = 10; // Initially an integer
variable = 'Hello';    // Now a string
```

✓ **Null:**

- Declare nullable variables using ?.

- Example:

```
String? nullableString; // A nullable string that can be null
```

✓ **Type Inference:**

- Use var, final, or const to let the type be inferred.
- Example:

```
var name = 'Alice'; // Inferred as String
final number = 42;  // Inferred as int
const isActive = true; // Inferred as bool
```

Step 3: Naming Conventions

✓ **General Guidelines:**

- Use **Camel Case** for variables, methods, and parameters.
  - Example: myVariable, calculateSum().
- Use **Pascal Case** for class and enum names.
  - Example: MyClass, AnimalType.
- Use **Snake Case** for constants (less common).
  - Example: MAX_LENGTH, API_KEY.

✓ **Naming Variables:**

- Use **descriptive names**.
  - Example: userName, totalPrice.
- Avoid abbreviations.
  - Example: Use customerAccount instead of custAcct.

✓ **Naming Methods:**

- Use **verb phrases** for method names.
  - Example: fetchData(), calculateTotal().
- For boolean methods, prefix with is, has, or can.
  - Example: isEmpty(), hasPermission().

✓ **Naming Classes and Enums:**

- Use nouns for class names.
  - Example: Car, UserProfile.
- Enum names should use Pascal case; values in uppercase snake case.

- Example:

```
enum AnimalType { dog, cat, bird }
```

✓ **Naming Constants:**

- Use uppercase letters with underscores for constants.
  - Example: MAX_CONNECTIONS, DEFAULT_TIMEOUT.

✓ **File and Directory Naming:**

- Use lowercase with underscores for file names.
  - Example: user_profile.dart, data_service.dart.

✓ **Package Naming:**

- Use all lowercase letters for package names.
  - Example: my_awesome_package.

Practical Activity 1.2.2: Applying control flow structures.

Task:

1. Read Key reading 1.2.2
2. Referring to the key reading 1.2.2, You are requested to go to the computer lab to apply control flow structures (conditional statements, sequence switch statement and iterative statements) in dart.
3. Present your work to the trainer/ your colleagues.
4. Ask for clarification if any.

Key readings 1.2.2: Applying control flow structures

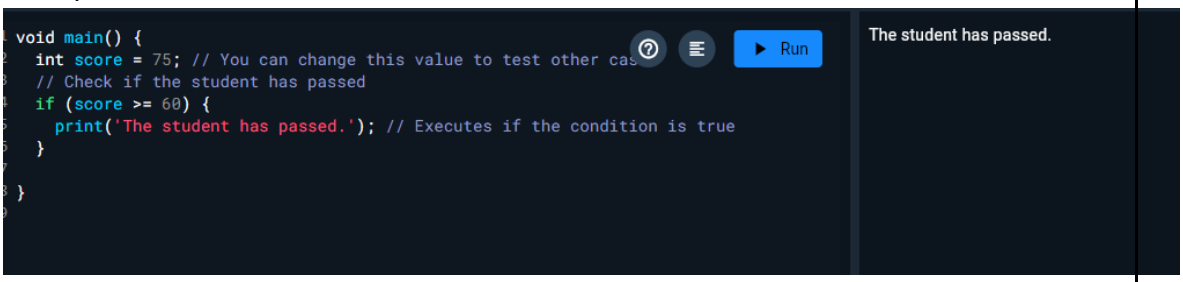
Here's a brief overview of control flow structures in Dart:

- **Conditional Statements**

1.if Statement: Executes a block of code if a specified condition is true.

```
if (condition) {
  // Code to execute if condition is true
}
```

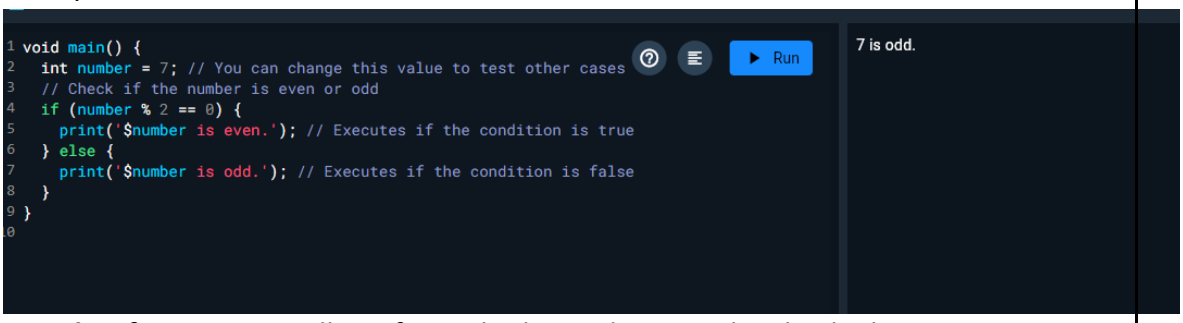
Example:



2.else Statement: Executes a block of code if the condition in the if statement is false.

```
if (condition) {
  // Code for true branch
} else {
  // Code for false branch
}
```

Example:



3.else if Statement: Allows for multiple conditions to be checked in sequence.

```
if (condition1) {
  // Code for condition1
} else if (condition2) {
  // Code for condition2
} else {
  // Code if neither condition is true
}
```

Example:

```dart
void main() {
  int score = 85; // You can change this value to test other cases
  // Determine the letter grade based on the score
  if (score >= 90) {
    print('Grade: A'); // Executes if score is 90 or above
  } else if (score >= 80) {
    print('Grade: B'); // Executes if score is between 80 and 89
  } else if (score >= 70) {
    print('Grade: C'); // Executes if score is between 70 and 79
  } else if (score >= 60) {
    print('Grade: D'); // Executes if score is between 60 and 69
  } else {
    print('Grade: F'); // Executes if score is below 60
  }
}
```

Run

```
Grade: B
```

- **Sequence switch statements**

switch Statement: Evaluates an expression and executes code based on matching cases.

switch (value) {
case case1:
// Code for case1
break;
case case2:
// Code for case2
break;
default:
// Code if no cases match
}

Example:

```dart
void main() {
  int day = 3; // You can change this value to test other cases

  // Determine the day of the week based on the integer
  switch (day) {
    case 1:
      print('Monday');
      break; // Exit the switch statement
    case 2:
      print('Tuesday');
      break;
    case 3:
      print('Wednesday');
      break;
    case 4:
      print('Thursday');
      break;
    case 5:
      print('Friday');
      break;
    case 6:
      print('Saturday');
      break;
    case 7:
      print('Sunday');
      break;
    default:
      print('Invalid day'); // Executes if no cases match
  }
}
```

Run

```
Wednesday
```

Iterating statements

Iterating statements, often referred to as loops, are fundamental control flow structures in programming that allow you to repeatedly execute a block of code as long as a specified condition is true.

In Dart, there are several types of iterating statements:

1. for Loop: Repeats a block of code a specified number of times.

Syntax:

```
for (initialization; condition; increment) {
  // Code to execute on each iteration
}
```

Example:

```dart
void main() {
  int sum = 0; // Variable to store the sum
  // Using a for loop to calculate the sum
  for (int i = 1; i <= 10; i++) {
    sum += i; // Add the current number to the sum
  }
  // Print the result
  print('The sum of the first 10 natural numbers is: $sum');
}
```

Run

```
The sum of the first 10 natural numbers is: 55
```

2. while Loop: Repeats a block of code as long as a specified condition is true.

Syntax:

```
Initialization;
while (condition) {
  // Code to execute
}
```

Example:

```dart
void main() {
  int count = 1; // Initialize the count variable

  // Using a while loop to print numbers from 1 to 5
  while (count <= 5) {
    print(count); // Print the current value of count
    count++;      // Increment count by 1
  }
}
```

Run

```
1
2
3
4
5
```

3.do-while Loop: Similar to the while loop, but it guarantees that the block of code is executed at least once.

Syntax:

```
Initialization;
do {
  // Code to execute
} while (condition);
```

for-in Loop: Used to iterate over elements in a collection (like a list or set).

```
for (var item in collection) {
```

// Code to execute for each item
}

Example:

```
void main() {
  int count = 1; // Initialize the count variable
  // Using a do-while loop to print numbers from 1 to 5
  do {
    print(count); // Print the current value of count
    count++;      // Increment count by 1
  } while (count <= 5); // Condition checked after the loop body
}
```

Run

```
1
2
3
4
5
```

4.break: Exits the nearest enclosing loop or switch statement.

Example:

```
for (int i = 0; i < 10; i++) {
if (i == 5) break; // Exit the loop when i equals 5
}
```

5.continue: Skips the current iteration of a loop and proceeds to the next iteration.

Example:

```
for (int i = 0; i < 10; i++) {
if (i % 2 == 0) continue; // Skip even numbers
// Code for odd numbers
}
```

Practical Activity 1.2.3: Performing functions in dart

Task:

1: Read Key reading 1.2.3
2: Referring to the key reading 1.2.3, You are requested to go to the computer lab to perform functions in dart.
3: Present your work to the trainer/ your colleagues.
4: Ask for clarification if any.

Key readings 1.2.3: Performing functions in dart

Here are the implementation steps for using built-in functions, declaring your own functions, specifying parameters and return types, and calling functions in Dart:

Step 1: Using Built-in Functions

1. Printing Output:

- Use the **print()** function to display output in the console.

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

2. Mathematical Operations:

- Import the dart: math library to access mathematical functions.

```
import 'dart:math';

void main() {
  print(max(10, 20));  // Prints: 20
  print(sqrt(16));     // Prints: 4.0
}
```

3. String Manipulation:

- Use built-in string functions to manipulate text.

```
void main() {
  String text = 'Dart Programming';
  print(text.toUpperCase());   // Prints: DART PROGRAMMING
  print(text.substring(0, 4)); // Prints: Dart
  print(text.contains('Program')); // Prints: true
}
```

4. Converting Data Types:

- Convert strings to numbers and vice versa.

```
void main() {
  List<int> numbers = [1, 2, 3];
  numbers.add(4);      // Adds 4 to the list
  numbers.remove(2);   // Removes 2 from the list
  print(numbers);      // Prints: [1, 3, 4]
  print(numbers.length); // Prints the number of items: 3
}
```

Step 2: Declaring Functions

Syntax for Declaring Functions:

```
ReturnType functionName(ParameterType parameter1, ParameterType parameter2) {
  // Function body
  return value;
}
```

Example:

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

Step 3: Perform parameters and return types

- Specify the parameter types and the return type in your function declaration.

Example:

```
int multiply(int a, int b) {
  return a * b;
}
```

Step 4: Calling functions

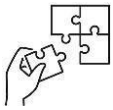
- Invoke your functions by passing the necessary arguments.

Example:

```
void main() {
  int result = add(5, 3);  // Calling the function with arguments 5 and 3
  print(result);           // Prints: 8
}
```

Points to Remember

- **While performing variable declaration, pass through the following steps:**
  - ✓ Using var, Using final and Using const.
  - ✓ Using datatypes.
  - ✓ Naming variable
- **While applying control flow structures pass through the following steps:**
  - ✓ Use Conditional Statements:Implement if, else to control flow based on conditions.
  - ✓ Sequence switch statements
  - ✓ Iterating statements: Use for loop, while loop, and do-while loop, break and continue.
- **While Performing functions pass through the following steps:**
  - ✓ Using built-in functions
  - ✓ Declaring functions
  - ✓ Perform parameters and return types.
  - ✓ Calling functions.



Application of learning 1.2.

DevSolutions Inc needs to create a simple mobile application that helps employees manage their tasks. The app will allow users to add tasks, mark them as complete, and view their progress. As mobile developer you are requested to perform variable declaration, apply control flow structures and functions in dart.

Indicative content 1.3: Applying Object Oriented Programming (OOP)

Duration: 5 hrs

Theoretical Activity 1.3.1: Description of Object-Oriented Programming (OOP)

Tasks:

1: You are requested to answer the following questions:
   i. Explain the following terms:
      1. Classes
      2. Objects
      3. Inheritance
      4. Polymorphism
      5. Encapsulation
      6. Abstraction
2: Write your findings to flipchart/papers
3: Present the findings to the whole class or trainer
4: For more clarification, read the key readings 1.3.1. In addition, ask questions whe re necessary.

Key readings 1.3.1: Description of Object-Oriented Programming (OOP)

This will breakdown of each of these concepts in dart:

1. Classes

A **class** is a blueprint or template for creating objects (instances). It defines a set of attributes (variables) and behaviors (methods) that the objects created from the class will have. In Dart, classes provide structure and a way to model real-world entities.

Example:

```
class Car {
String model;
int year;
// Constructor
Car(this.model, this.year);
// Method
void displayInfo() {
print('Model: $model, Year: $year');
```

```
  }
}
```

2. Objects

An **object** is an instance of a class. When a class is defined, no memory is allocated until an object of that class is created. Objects hold the data (attributes) and allow access to methods defined in the class.

Example:

```
void main() {
Car myCar = Car('Tesla', 2022);  // Object of class Car
myCar.displayInfo();  // Accessing a method through the object
}
```

3. Inheritance

Inheritance allows one class (child or subclass) to inherit the properties (attributes and methods) of another class (parent or superclass). It helps in code reuse and organizing classes hierarchically. In Dart, the extends keyword is used for inheritance.

Example:

```
class Vehicle {
void start() {
print('Vehicle started');
  }
}

class Car extends Vehicle {
// Car inherits the start method from Vehicle
void honk() {
print('Car is honking');
  }
}

void main() {
  Car myCar = Car();
  myCar.start();  // Inherited from Vehicle
  myCar.honk();   // Defined in Car
}
```

4. Polymorphism

Polymorphism means "many forms." In object-oriented programming, it allows a single interface to represent different data types or classes. This means that a method or property can behave differently based on the object that is invoking it. In Dart, polymorphism is achieved through method overriding.

Example:

```
class Animal {
  void sound() {
    print('Animal makes a sound');
  }
}

class Dog extends Animal {
  @override
  void sound() {
    print('Dog barks');
  }
}

class Cat extends Animal {
  @override
  void sound() {
    print('Cat meows');
  }
}

void main() {
  Animal myDog = Dog();
  Animal myCat = Cat();

  myDog.sound();  // Dog barks
  myCat.sound();  // Cat meows
}
```

5. Encapsulation

Encapsulation is the practice of hiding the internal details or data of an object from the outside world and only exposing necessary parts via methods. This is typically done using access modifiers like private (by prefixing _ to variable names in Dart) to restrict direct access.

Example:

```
class BankAccount {
  double _balance;  // Private variable

  BankAccount(this._balance);

  // Public method to get balance (read-only access)
```

```
  double getBalance() {
    return _balance;
  }

  // Public method to deposit money (controlled access)
  void deposit(double amount) {
    if (amount > 0) {
      _balance += amount;
    }
  }
}
```

6. Abstraction

Abstraction is the concept of hiding complex details and showing only the essential features of an object. In Dart, this can be achieved using abstract classes and interfaces. An abstract class can define methods without an implementation, forcing subclasses to provide the concrete implementation.

Example:

```
abstract class Shape {
  void draw();  // Abstract method
}

class Circle extends Shape {
  @override
  void draw() {
    print('Drawing a circle');
  }
}

class Square extends Shape {
  @override
  void draw() {
    print('Drawing a square');
  }
}

void main() {
  Shape myCircle = Circle();
  Shape mySquare = Square();

  myCircle.draw();  // Drawing a circle
```

```
  mySquare.draw();  // Drawing a square
}
```

Note That:

- **Classes** define the structure of objects.
- **Objects** are instances of classes.
- **Inheritance** enables classes to inherit properties and behaviors from other classes.
- **Polymorphism** allows objects of different types to be treated as instances of the same class through a common interface.
- **Encapsulation** protects data by controlling access to it via methods.
- **Abstraction** hides unnecessary details and exposes only the essential features.

Practical Activity 1.3.2: Applying Object Oriented Programming in Dart

Task:

1: Read Key reading 1.3.2
2: Referring to the key reading 1.3.2, You are requested to go to the computer lab to apply the OOP in dart programming language.
3: Present your work to the trainer/ your colleagues.
4: Ask for clarification if any.

Key readings 1.3.2 Applying Object Oriented Programming in Dart

There are the steps to implement Object-Oriented Programming (OOP) concepts in Dart, including classes and objects, encapsulation, inheritance, polymorphism, and abstraction:

Step 1: Creating classes and objects

1. Create a Class:

- Create a blueprint using the class keyword.
- Define properties and methods within the class.

Example:

```dart
class Car {
  String brand;
  String model;

  Car(this.brand, this.model); // Constructor

  void displayInfo() {
    print('Car brand: $brand, model: $model');
  }
}
```

2. Create an Object:

- Instantiate an object (instance) of the class using the new keyword or directly.

Example:

```dart
void main() {
  Car myCar = Car('Toyota', 'Corolla'); // Creating an object
  myCar.displayInfo(); // Calls the method
}
```

Step 2: Creating Encapsulation

1. Bundle Data and Methods:

- Use private variables (by prefixing with _) and public methods to access them.

Example:

```dart
class BankAccount {
  String _accountNumber; // Private variable
  double _balance;

  BankAccount(this._accountNumber, this._balance);

  void deposit(double amount) {
    _balance += amount;
  }

  void withdraw(double amount) {
    if (amount <= _balance) {
      _balance -= amount;
    } else {
      print('Insufficient funds');
    }
  }

  double get balance => _balance; // Getter for balance
}
```

Step 3: Creating Inheritance

1. Create a Base Class:

- Define a class that will serve as a parent class.

Example:

```dart
class Vehicle {
  void start() {
    print('Vehicle started');
  }
}
```

2. Create a Derived Class:

- Use the extends keyword to inherit from the base class.

Example:

```dart
class Motorcycle extends Vehicle {
  void revEngine() {
    print('Motorcycle engine revved');
  }
}
```

3. Use Inherited Methods:

```dart
void main() {
  Motorcycle myBike = Motorcycle();
  myBike.start(); // Inherited method
  myBike.revEngine(); // Specific method
}
```

Step 4: Creating Polymorphism

1. Define Methods in Base Class:

- Create a method in the base class that can be overridden.

Example:

```dart
class Animal {
  void sound() {
    print('Animal sound');
  }
}
```

2.Override in Derived Classes:

```dart
class Dog extends Animal {
  @override
  void sound() {
    print('Bark');
  }
}

class Cat extends Animal {
  @override
  void sound() {
    print('Meow');
  }
}
```

3. Use Polymorphism:

```dart
void main() {
  Animal myDog = Dog();
  Animal myCat = Cat();

  myDog.sound(); // Prints: Bark
  myCat.sound(); // Prints: Meow
}
```

Step 5:Creating abstract

1. Define an Abstract Class:

- Use the abstract keyword to create a class that cannot be instantiated directly.

Example:

```dart
abstract class Shape {
  double area(); // Abstract method
}
```

2. Implement the Abstract Class:

- Create subclasses that implement the abstract methods.

Example:

```dart
class Rectangle extends Shape {
  double width;
  double height;

  Rectangle(this.width, this.height);

  @override
  double area() {
    return width * height; // Implementation
  }
}

class Circle extends Shape {
  double radius;

  Circle(this.radius);

  @override
  double area() {
    return 3.14 * radius * radius; // Implementation
  }
}
```

3. Use Abstraction:

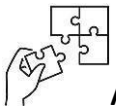
```dart
void main() {
  Shape rectangle = Rectangle(5, 10);
  Shape circle = Circle(7);

  print('Rectangle area: ${rectangle.area()}'); // Prints: Rectangle area: 50.0
  print('Circle area: ${circle.area()}');       // Prints: Circle area: 153.86
}
```

Points to Remember

- Description of Object-Oriented Programming (OOP) terms:
  - ✓ **Class** is a blueprint for creating object.
  - ✓ **Object** is an instance of class.
  - ✓ **Encapsulation**Bundle data and methods, restricting access.
  - ✓ **Inheritance** is a fundamental concept in object-oriented programming (OOP) that allows a class to inherit properties and methods from another class

- ✓ **Polymorphism**: Polymorphism is a core principle of object-oriented programming that allows methods to do different things based on the object that it is acting upon.
- ✓ **Abstraction**is a fundamental concept in object-oriented programming (OOP) that involves hiding complex implementation details and exposing only the necessary parts of an object or system.

- While apply Object Oriented Programming, pass through the following steps:
  - ✓ Create class and object.
  - ✓ Create Inheritance
  - ✓ Create Polymorphism
  - ✓ Create Encapsulation
  - ✓ Create Abstraction



Application of learning 1.3.

Tech Innovations Inc manages different types of employees (e.g., full-time, part-time, and interns) and their specific attributes and behaviours. The system will allow HR personnel to add employees, calculate salaries, and generate reports based on employee types. As mobile developer, you are requested to create classes and objects, inheritance, polymorphism, encapsulation, and abstraction.

Indicative content 1.4: Using Dart Libraries and Packages

Duration: 5 hrs

Theoretical Activity 1.4.1: Description of dart libraries and packages

Tasks:

1: You are requested to answer the following questions:
   a. Discuss the following dart libraries and packagies:
   b. Importing and using libraries
   c. Exploring built-in Dart libraries
   d. Managing dependencies with pub (Dart's package manager)
   e. Using external packages for enhanced functionality
2: Involve trainees in presentation of their findings.
3: Provide expert view and clarifies ideas by using didactic materials.
4: Address any questions or concerns.
5: Ask trainees to read the key reading 1.4.1 in the trainee manual.

Key readings 1.4.1: Description of dart libraries and packages

✓ **Importing and using libraries**

Use the **import** keyword to bring in built-in or third-party libraries.

Example:

```
import 'dart:math';
void main() {
  print(pi);  // Prints the value of Pi from the 'dart:math' library.
}
```

✓ **Exploring built-in Dart libraries**

- dart:core: Core library for basic Dart functionality (automatically imported).
- dart:async: For asynchronous programming, such as using *Future* and *Stream*.
- dart:convert: For encoding and decoding JSON, UTF-8, and other conversions

✓ **Managing dependencies with pub (Dart's package manager)**

pubspec.yaml: This file is where dependencies and other project details are listed.

Example: Adding a Dependency:

```
dependencies:
  http: ^0.13.0
```

This adds the http package to your Dart project, which can be used for making

HTTP requests. To install the dependencies, run dart pub get in your project directory.

✓ **Using external packages for enhanced functionality**

1. **Find a package** on pub.dev (e.g., http for HTTP requests).
2. **Add the package** to your pubspec.yaml file:
   dependencies:
   http: ^0.13.0
3. **Install the package** by running dart pub get.
4. **Import the package** into your code:
   import 'package:http/http.dart' as http;
5. **Use the package** to add functionality to your app:
   var response = await http.get(Uri.parse('https://example.com'));
   print(response.body);

Practical Activity 1.4.2: Performing dart libraries and packages

Task:

1: Read Key reading 1.4.2
2: Referring to the key reading 1.4.2, You are requested to go to the computer lab to perform dart libraries and packages
3: Present your work to the trainer/ your colleagues.
4: Ask for clarification if any.

Key readings 1.4.2Performingdart libraries and packages

You are now in the computer lab. Your task is to import built-in Dart libraries and external packages into your project. Follow the steps below:

Step 1: Importing Built-In Libraries

Dart comes with several built-in libraries. Let's import one of the most common ones, such as **dart: math** for mathematical functions.

Example:

```
import 'dart:math';
void main() {
 var randomNumber = Random().nextInt(100); // Generates a random number between 0 and 99
 print('Random number: $randomNumber');
}
```

Step 2: Importing External Packages

To use external packages, you need to add them to your project. Here's how you do it:

1. Open your **pubspec.yaml** file.
2. Add the external package (for example, http for making HTTP requests).

pubspec.yaml:

```
dependencies:
  http: ^0.14.0
```

3. Run **flutter pub get** or **dart pub get** to install the package.
4. Now, you can import the package in your Dart/Flutter file.

Example:

```
import 'package:http/http.dart' as http;

void main() async {
  var response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts/1'));
  print('Response status: ${response.statusCode}');
  print('Response body: ${response.body}');
}
```

Task Instructions:

I. Open the **pubspec.yaml** file in your project and add an external package of your choice **(e.g., http, provider).**

II. Import a built-in Dart library **(dart:math, dart:io, etc**.) and demonstrate how to use its functionality.

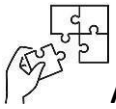
III. Use the external package you added in a basic program (e.g., make a simple HTTP request using the http package).

Points to Remember

Description of using dart libraries and packages

- ✓ Importing and using libraries
- ✓ Exploring built-in Dart libraries
- ✓ Managing dependencies with pub (Dart's package manager)
- ✓ Using external packages for enhanced functionality

- **While Performing dart libraries and packages, pass through the following steps:**
  - ✓ Importing Built-In Libraries
  - ✓ Importing External Packages



Application of learning 1.4.

WeatherWise Solutions is a tech start-up dedicated to providing accurate and timely weather information to users. Their goal is to develop an application that aggregates weather data from various sources, allowing users to access it easily for planning their activities. As a mobile developer, you are requested to set up a dart project for importing built-in libraries and adding external packages to fetch data from an API and store the results locally.

Learning outcome 1 end assessment

Theoretical assessment:

I. Circle the letter corresponding to the correct answer:

Question 1: What type of programming paradigm does Dart primarily support?

A) Functional programming

B) Object-oriented programming

C) Procedural programming

D) Logic programming

Question 2: Which of the following features is NOT a characteristic of Dart?

A) Strongly typed

B) Just-in-time (JIT) compilation

C) Automatic memory management

D) Lack of null safety

Question 3: Dart Frameworks

Which framework is commonly used with Dart for building web applications?

A) Angular

B) React

C) Flutter

D) Vue.js

Question 4: Dart is primarily used for which of the following applications?

A) Desktop applications only

B) Mobile applications only

C) Web and mobile applications

D) Game development only

Question 5: Which feature of Dart enhances its performance for mobile applications?

A) Dart VM

B) Hot reload

C) Dynamic typing

D) Synchronous programming

Question 6: What is the primary framework for building cross-platform mobile applications using Dart?

A) AngularDart

B) Flutter

C) DartPad

D) Express.js

II. Answer by TRUE to the correct statements and FALSE to the wrong statements.

1) Dart supports both static and dynamic data types, allowing for flexibility in variable declaration.
2) In Dart, variables must be declared with a type before they can be used.....

3) Dart provides standard control flow structures like if, else, for, and switch.....
4) In Dart, functions are first-class citizens, meaning they can be assigned to variables and passed as arguments.
5) Dart can only be used for developing web applications and is not suitable for native app development.

III. Fill the empty by place by using the correct world listed in blacket.

(Class, Encapsulation, Polymorphism, Inheritance, Abstraction, Object).

A)In Dart, a blueprint for creating objects is called a _____.

B)_____ allows a class to inherit properties and methods from another class, promoting code reusability.

C)_____ is the ability of different classes to be treated as instances of the same class through a common interface.

D)_____ is a principle that restricts access to certain components of an object, allowing only controlled access through methods.

E)_____ is the concept of hiding complex implementation details and exposing only the necessary features of an object.

IV. Match the following terms to their description and write letters in empty place of answers.

Answers	**Term**	**Description**
1…….	1) Using built-in functions	A) Defining a function with specific inputs and outputs.
2……	2) Declaring functions	B) The act of executing a function's code.
3……..	3) Parameters and return types	C) Classification that specifies which type of value a variable can hold.
4……	4) Calling functions	D) Specifying the types of inputs and outputs in a function.
		C) Utilizing functions provided by Dart's libraries.

Practical Assessment

SmartHome Innovations is a technology company focused on developing smart home solutions that allow users to control and monitor their home devices through a centralized application. As a mobile developer you are requested to set up key development tools, apply dart concepts, apply Object-Oriented Programming (OOP), perform dart libraries and packages.

References

https://chatgpt.com/g/g-RGr8YRENd-chart-gpt-3

https://stackoverflow.com/

https://www.geeksforgeeks.org/what-is-ide/

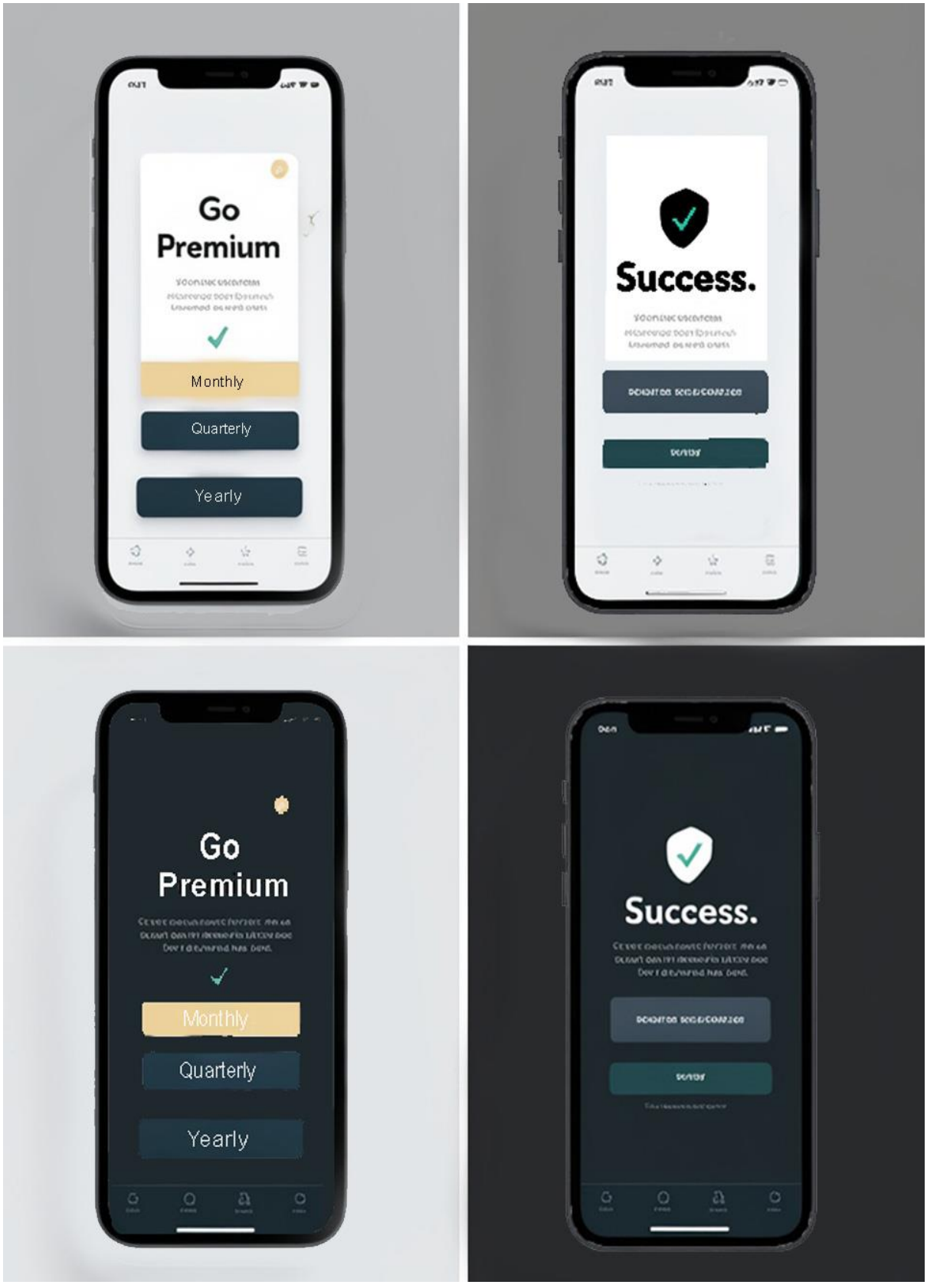
https://www.w3schools.com/

Rose, R. (2022). Flutter and Dart Cookbook 1st Edition. Calfornia: O'Reilly Media.

Thornton, E. (2020). Flutter For Beginners: A Genius Guide to Flutter App Development. KDP: Amazon Digital Services LLC.

Tutorials Point. (2019). Flutter. Retrieved from Tutorials Point (I) Pvt. Ltd:

.


Go
Premium
Monthly
Quarterly
Yearly
Success.
Go
Premium
Monthly
Quarterly
Yearly
Success.

Indicative Contents
2.1 Preparation of Flutter Environment
2.2 Applying Flutter's Widget System
2.3 Implementation of State Management
2.4 Using Pre-Designed Widgets

Key Competencies for Learning Outcome 2: Implement UI designs

Knowledge	Skills	Attitudes
● Description of flutter environment ● Description of flutter's widget system. ● Description of state management. ● Description of pre-designed widgets	● Preparing Flutter Environment ● Applying flutter's widgets design. ● Implementing state management ● Using pre-designed widgets	● Having a teamwork spirit. ● Being creative in Optimizing Performance ● Being adaptive. ● Having Time management ability ● Being critical thinker ● Being Innovative in Unique Design Solutions ● Having Passion for Learning

Duration: 30 hrs

Learning outcome 2 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe correctly flutter framework as applied in software development
2. Describe correctly widgets as used in flutter software development.
3. Describe correctly state management as used in flutter software development.
4. Install correctly Flutter SDK, IDE (Android Studio) and Configuration of development environment as used in mobile application development.
5. Create properly a new Flutter project according to development environment.
6. Develop correctly flutter's widget system based on flutter functionalities.
7. Implement state management based on flutter functionalities.
8. Use correctly pre-designed widgets as required in flutter framework.

Resources

Equipment	Tools	Materials
● Computer (Windows, MacOS) ● Mobile Phone (Android, iPhone)	● VS Code ● Android Studio ● Xcode ● Emulators ● Flutter SDK. ● FlutterFlow ● Figma	● Flutter Icons ● Internet

Indicative content 2.1: Preparation of flutter environment

Duration: 7 hrs

Theoretical Activity 2.1.1: Introduction to Flutter framework

Tasks:

Step 1: You are requested to answer the following questions:

I. Define the following terms:
   a. Flutter
   b. Widgets
   c. State management
   d. core widgets
   e. Widget Tree

ii. What are purposes of flutter framework?

iii. Identify features of flutter framework.

iv. Differentiate types of widgets

v. Differentiate widget Lifecycle from packages used in flutter framework.

vi. What are methods and Libraries used in flutter state management?

Step 2: Involve trainees in presentation of their findings

Step 3: Provide expert view and clarifies ideas by using Flutter Widgets.

Step 4: Address any questions or concerns.

Step 5: Ask trainees to read the key reading 2.1.1 in the trainee manual.

Key readings 2.1.1Introduction to Flutter Framework

- **Definition**

  Flutter is an open-source UI software development kit (SDK) created by Google for building natively compiled applications across multiple platforms, including mobile (iOS, Android), web, and desktop (Windows, macOS, Linux) from a single codebase. It uses the Dart programming language to create fast, expressive, and flexible user interfaces.

- **Purpose**

  The primary purpose of Flutter is to enable developers to create high-performance, visually attractive applications that work seamlessly across different platforms using a single codebase. This minimizes development effort

and allows for faster development cycles while ensuring a native-like experience on each platform.

- **Features of Flutter**

1. **Cross-platform Development**: Write once and deploy to multiple platforms (iOS, Android, web, desktop).
2. **Hot Reload**: Instantly view the effects of changes in the code without restarting the app. This speeds up the development process.
3. **Single Codebase**: One codebase for multiple platforms reduces redundancy and maintenance overhead.
4. **Customizable Widgets**: Flutter provides a rich library of customizable widgets that allow for the creation of complex UIs.
5. **Native Performance**: Flutter compiles to ARM or Intel machine code, ensuring high performance and native-like speed on mobile platforms.
6. **Expressive UI**: With Flutter's flexible layout and powerful compositing capabilities, developers can easily create beautiful, responsive UIs.
7. **Dart Language**: Powered by Dart, a fast and modern programming language with strong support for async programming and concise syntax.
8. **Rich Ecosystem**: Flutter comes with an extensive package ecosystem, providing tools for networking, database access, state management, and more.
9. **Declarative UI**: UI is built declaratively in Flutter, which makes code more readable and easier to manage.
10. **Support for Animation**: Flutter has built-in support for creating complex animations with ease.
11. **Open Source**: Being open-source, Flutter is supported by a large and active community, along with regular updates and contributions from Google.

✓ **Widgets in Flutter**

- **Definition**

Widgets are the core building blocks of a Flutter application. They describe the structure, layout, and behavior of UI elements. Every visual component in Flutter is a widget, including text, images, buttons, layouts, and even the entire screen. Flutter uses a composition-based approach, meaning complex UIs are created by nesting smaller widgets.

- **Types of Widgets**

Widgets in Flutter are broadly categorized into two types:

1. **Stateless Widgets**:

- **Definition**: Stateless widgets do not store any state information. Their appearance and behavior are based solely on the input properties passed to them. They are immutable, meaning once they are built, they cannot be changed.
- **Example**: Text, Icon, and Image widgets are stateless.

- **Use Case**: For UI elements that do not require any dynamic change, such as static text or images.

```
class MyStatelessWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Text("Hello, Flutter!");
  }
}
```

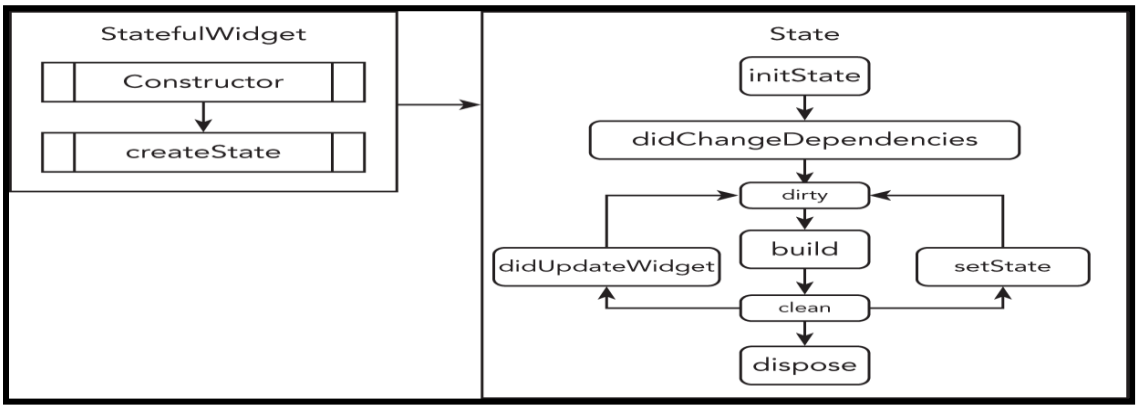
2. **Stateful Widgets**:
   - **Definition**: Stateful widgets are dynamic and can change over time in response to user interaction or other events. They have an associated state object that holds information that can be updated.
   - **Example**: Checkbox, Slider, and Forms are examples of stateful widgets.
   - **Use Case**: For UI elements that need to update or redraw based on user interaction or external data (e.g., a button click changing the UI).

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

class _MyStatefulWidgetState extends State<MyStatefulWidget> {
  String text = "Hello!";

  @override
  Widget build(BuildContext context) {
  return Column(
  children: [
  Text(text),
  ElevatedButton(
    onPressed: () {
 setState(() {
  text = "Button Pressed!";
 });

child: Text("Press me"),
)
],
   );
```

```
  }
}
```

- **Widget Lifecycle**

  The lifecycle of a widget is especially important for **StatefulWidgets**, as it defines the stages of a widget's existence from its creation to its removal.



The key stages of the widget lifecycle are:

1. **createState()**:
   - This is the first method called when a stateful widget is created. It creates an instance of the widget's state object (State).

```
@override
_MyWidgetState createState() => _MyWidgetState();
```

2. **initState()**:
   - This method is called once, immediately after the widget is inserted into the widget tree. It's used for any initial setup, such as subscribing to a service or initializing data.

```
@override
void initState() {
  super.initState();
  // Initialization code here
}
```

3. **build()**:
   - The build() method is called whenever the widget needs to be rendered. This method can be called multiple times during the widget's lifecycle as a result of state changes or updates from the framework.

```
@override
Widget build(BuildContext context) {
  return Text("Building UI");
```

```
}
```

4. **didUpdateWidget()**:
   - Called whenever the widget is rebuilt due to changes in the parent widget. This method provides an opportunity to respond to these changes, such as updating dependent data.

```
@override
void didUpdateWidget(MyWidget oldWidget) {
  super.didUpdateWidget(oldWidget);
  // Handle changes
}
```

5. **setState()**:
   - This method is called to inform Flutter that the widget's state has changed and it needs to rebuild. It triggers the build() method again to update the UI.

```
void updateText() {
  setState(() {
    text = "New Text!";
  });
}
```

6. **deactivate()**:
   - This is called when the widget is removed from the widget tree but may be reinserted before being disposed of. It's used to release resources or cancel subscriptions temporarily.

```
@override
void deactivate() {
  super.deactivate();
  // Clean up or pause resources
}
```

7. **dispose()**:
   - This method is called when the widget is permanently removed from the tree and will not be rebuilt. It is used for cleanup, such as releasing resources or canceling subscriptions.

```
@override
void dispose() {
  // Clean up resources
  super.dispose();
}
```

✓ **State Management:**

Definition: State management refers to the way an application handles and maintains the state of its components, such as data and UI elements, across different interactions, user inputs, and events. It ensures that the UI reflects the current state of the application and changes dynamically in response to user actions or external events.

Packages:

Definition: A package is a bundle of code that is shared and distributed. It typically includes one or more libraries, tools, assets, and other resources that you can import into your project to add functionality. Packages simplify the reuse of code and are often maintained by the community or organizations.

Purpose: Packages provide additional features or tools that help in development, such as state management, networking, or UI components.

Example (Flutter):

- **GetX**, **Provider**, **Riverpod**, **Bloc** are all **packages** that include tools to manage state.
- **npm** packages in JavaScript like **axios** for HTTP requests or **lodash** for utility functions.

Libraries:

Definition: A library is a collection of pre-written code that provides specific functionality, such as a utility function, framework features, or an entire architecture. Libraries can be a part of a package but focus solely on the code and functionality.

Purpose: Libraries are used to perform common tasks without the need to write code from scratch. They are often more focused than packages and are typically part of a larger package ecosystem.

Example:

- **Redux** is a **library** that helps manage application state in JavaScript.
- **RxJS** is a reactive programming library in Angular for handling asynchronous data.
- **Lodash** is a JavaScript utility library to simplify working with arrays, numbers, objects, etc.

Methods:

- **Definition:** Methods refer to specific functions or techniques used within code to achieve a particular goal. They could be part of a class, object, or even a standalone function in programming languages. In the context of state management, methods are specific techniques or patterns used to manage state.
- **Purpose:** Methods perform actions or operations. In state management,

methods might refer to how you update the state, manage side effects, or handle user input.

- **Example:**
  - In **Redux**, methods like dispatch() send actions to reducers.
  - In **GetX**, methods like update() are used to notify the UI of state changes.
  - In a **StatefulWidget** in Flutter, methods like setState() are used to update the widget's state.

Note That:

Packages: are comprehensive bundles that can include multiple libraries and tools.

Libraries: are collections of code that focus on specific functionality.

Methods: are individual functions or techniques used to implement the logic.

Practical Activity 2.1.2: Preparation of flutter environment

Task:

1: Read Key reading 2.1.2

2: Referring to the key reading 2.1.2, You are requested to go to the computer lab to prepare flutter environment and create new flutter project.

3: Present your work to the trainer/ your colleagues.

4: Ask for clarification if any.

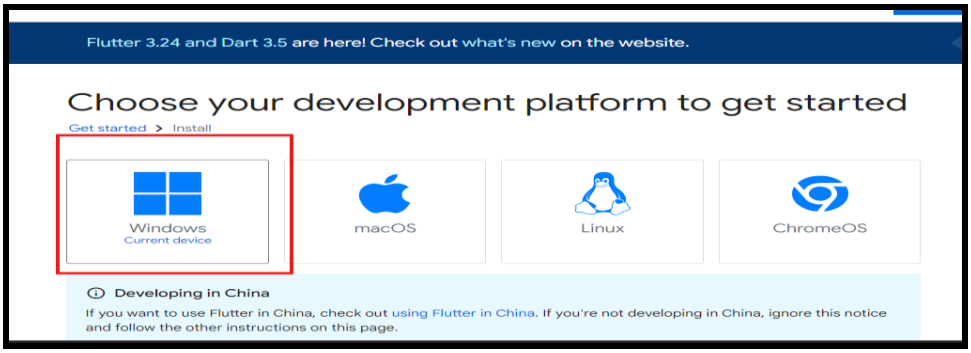
Key readings 2.1.2 Preparation of flutter environment:

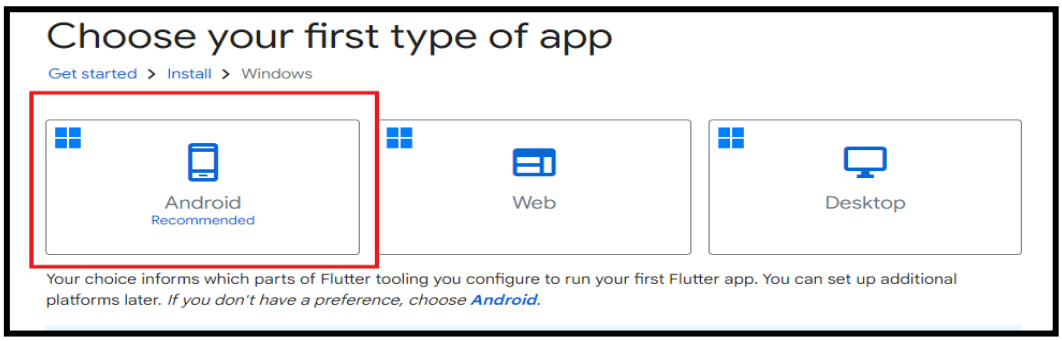
1. Install Flutter SDK

- Go to the official Flutter SDK website and follow the steps for your platform (Windows, macOS, or Linux).

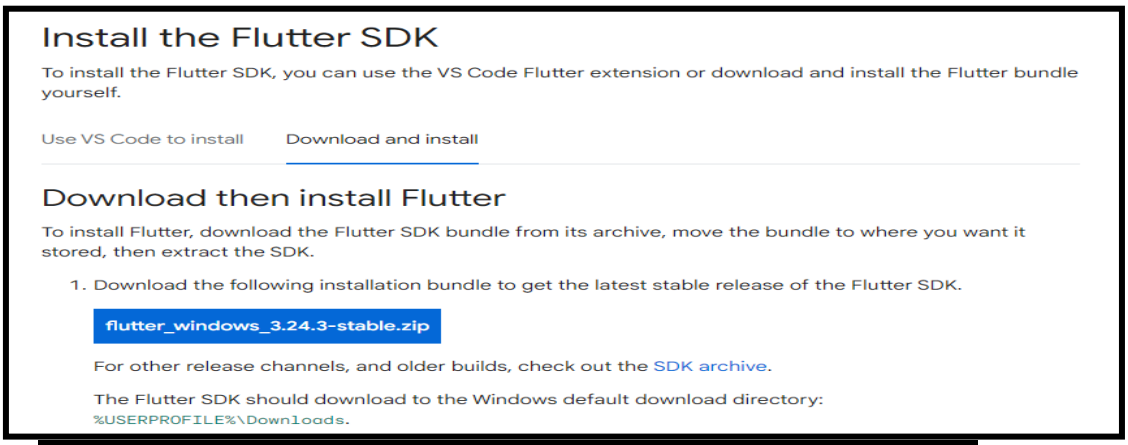
For Windows:

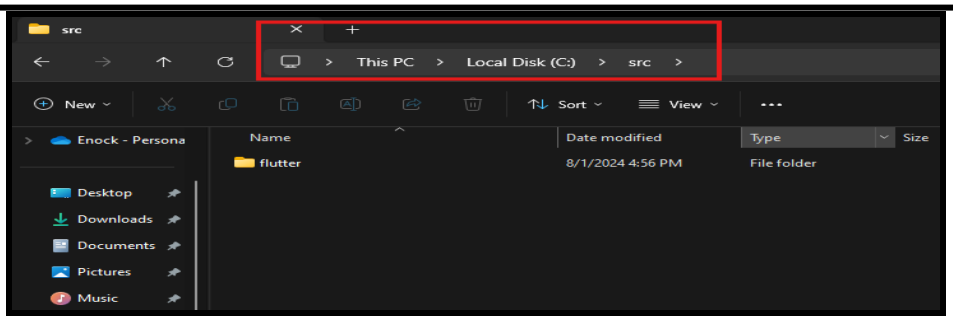
1. Download the latest Flutter SDK.





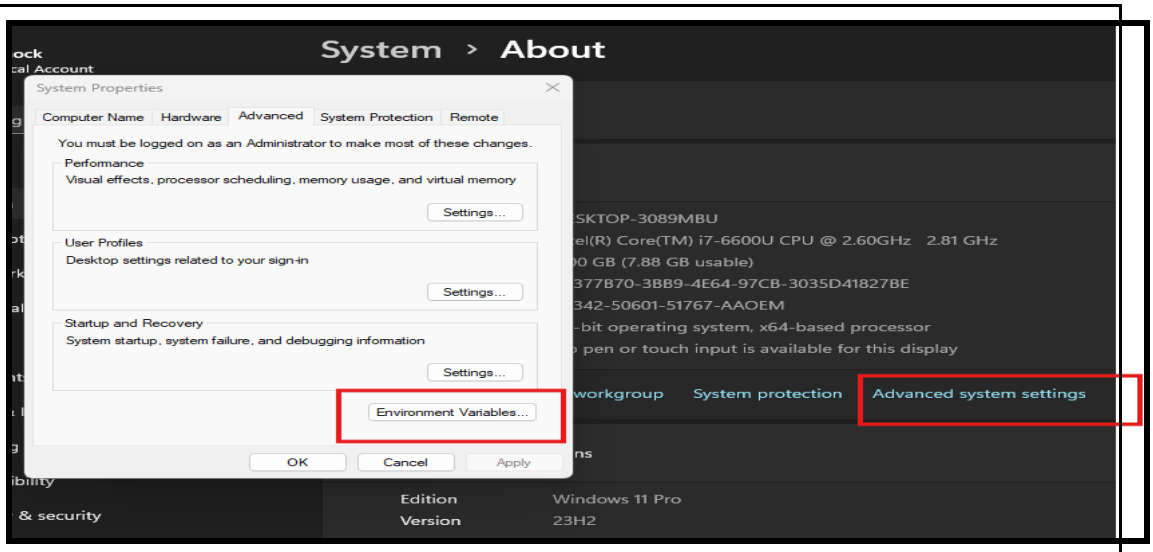
2. Extract the file to your desired location (e.g., C:\src\flutter).



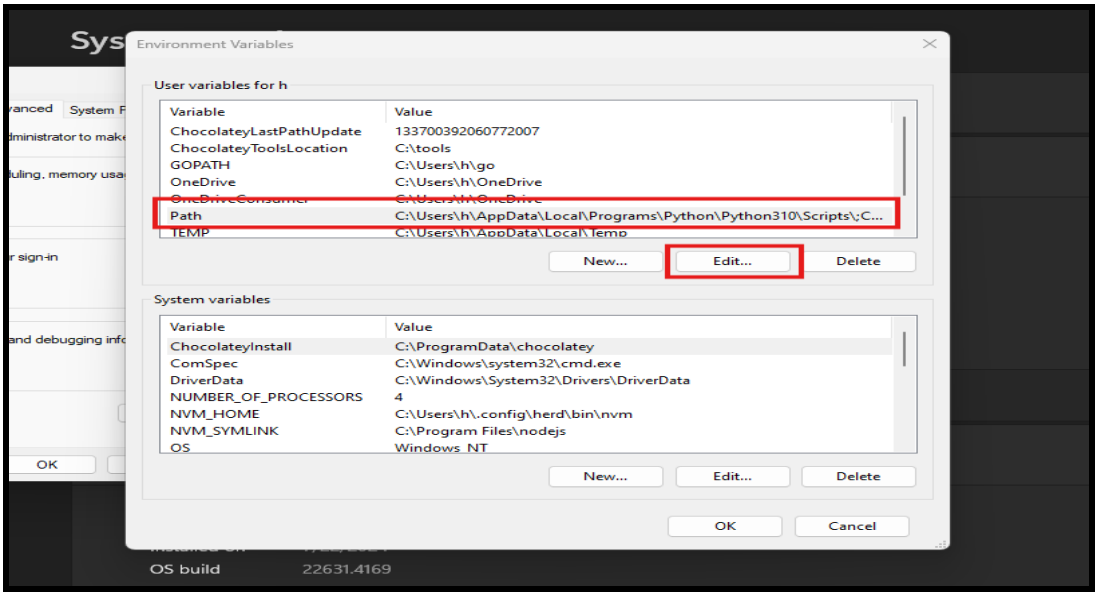


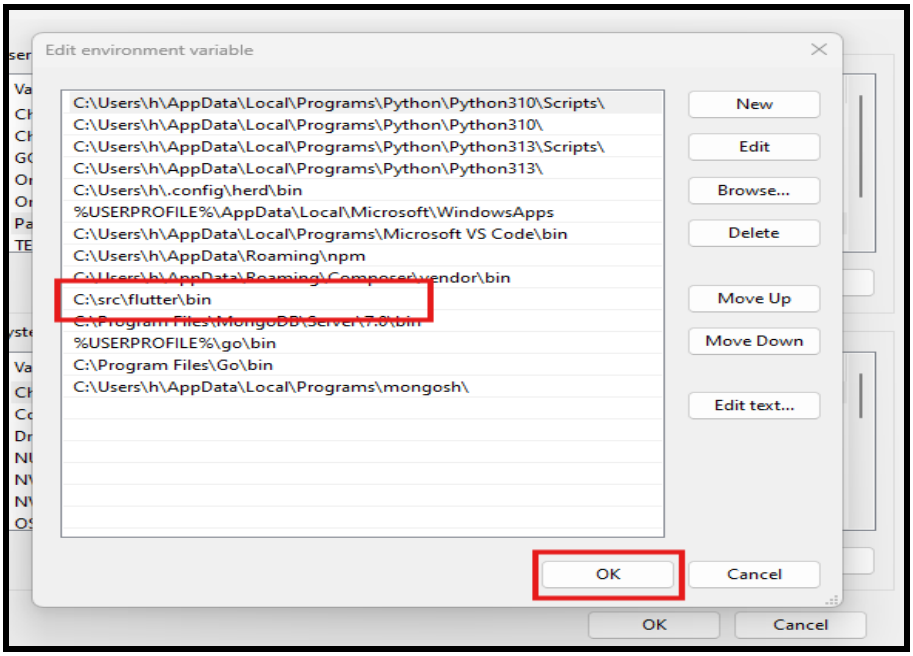
3. Add the Flutter bin directory to your system's PATH.

- Right-click on "This PC" > Properties > Advanced system settings > Environment Variables.



- Under "System Variables," select Path, click "Edit," and add the flutter\bin path.





For macOS:

1. Download the latest Flutter SDK from here.
2. Extract it and update your .bash_profile or .zshrc to include the flutter/bin path:

   export PATH="$PATH:`pwd`/flutter/bin"

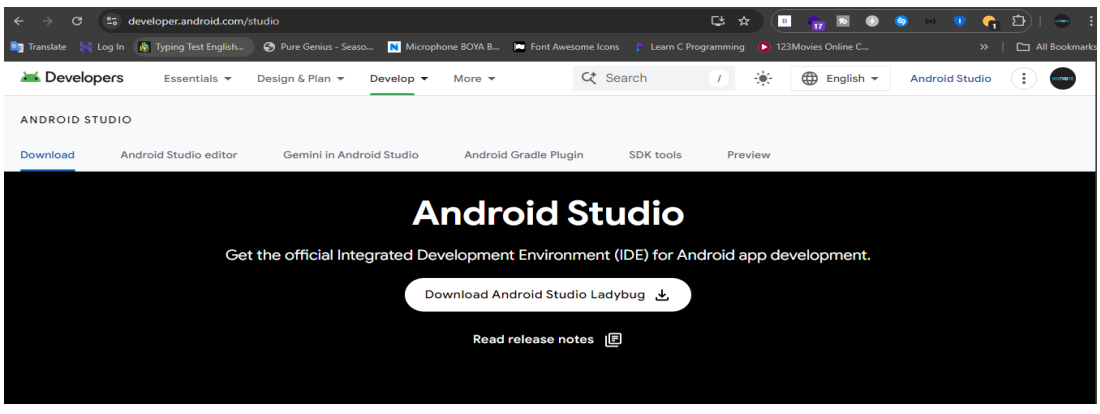
For Linux:

1. Download the Flutter SDK from here.
2. Extract it and add the flutter/bin directory to your PATH.

2. Install Android Studio for all platform or Xcode for macOS

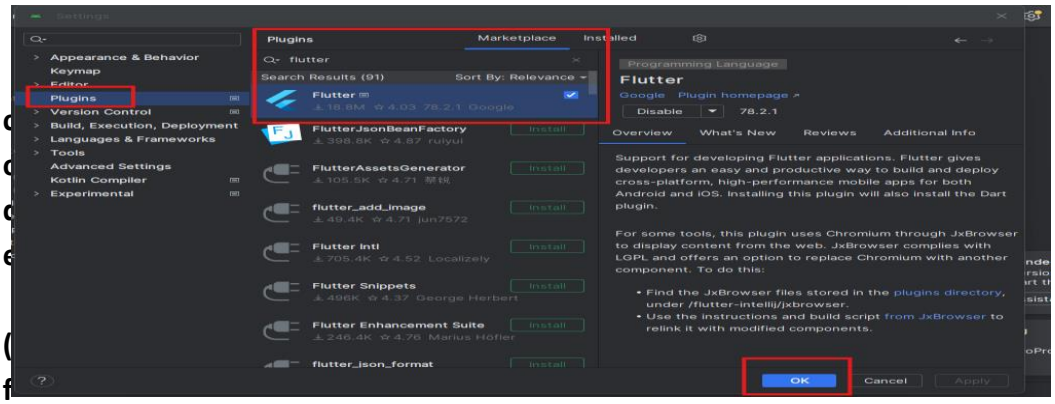
- **Android Studio:**

1. Download and install Android Studio from the official site



2. During installation, ensure you install the Android SDK, Android SDK Platform-Tools, and Android Virtual Device (AVD).

3. Open Android Studio and go to "Configure" > "Plugins" and install the **Flutter** plugin.

- 

or macOS only):

1. Install Xcode from the App Store.

2. Open Xcode and go to "Preferences" > "Locations" > Command Line Tools, and select the version of Xcode you're using.

3. Configure Flutter Development Environment

- **For Android:**

1. Open Android Studio and configure the AVD (Android Virtual Device) for running Flutter apps.

2. Open the terminal and run:

```
flutter doctor
```

- This command checks if your environment is set up correctly. Follow any prompts to fix missing dependencies.

- **For iOS (macOS only):**

1. Ensure Xcode is installed and configured.

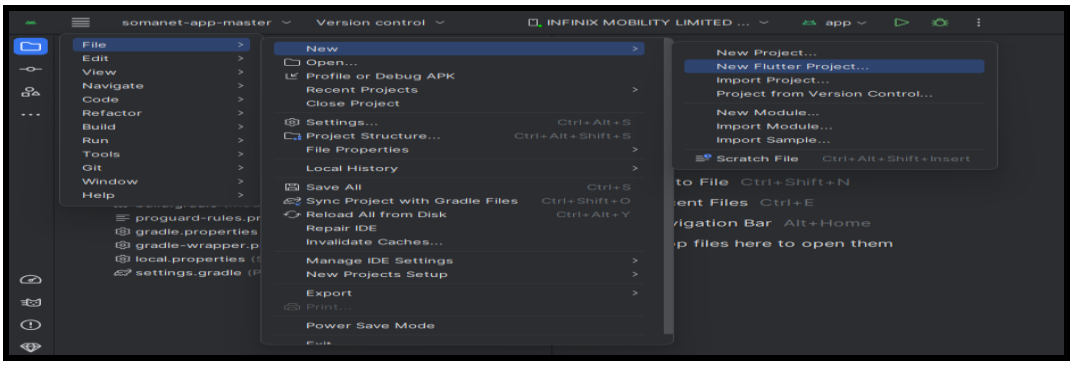
2. Run:

```
flutter doctor
```

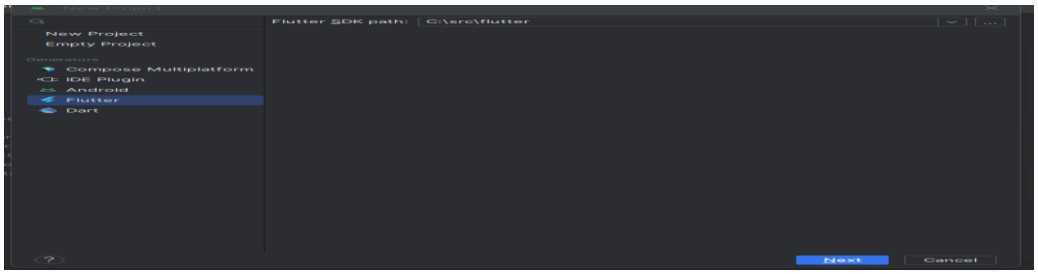
4. Create a New Flutter Project

1. Open Android Studio.

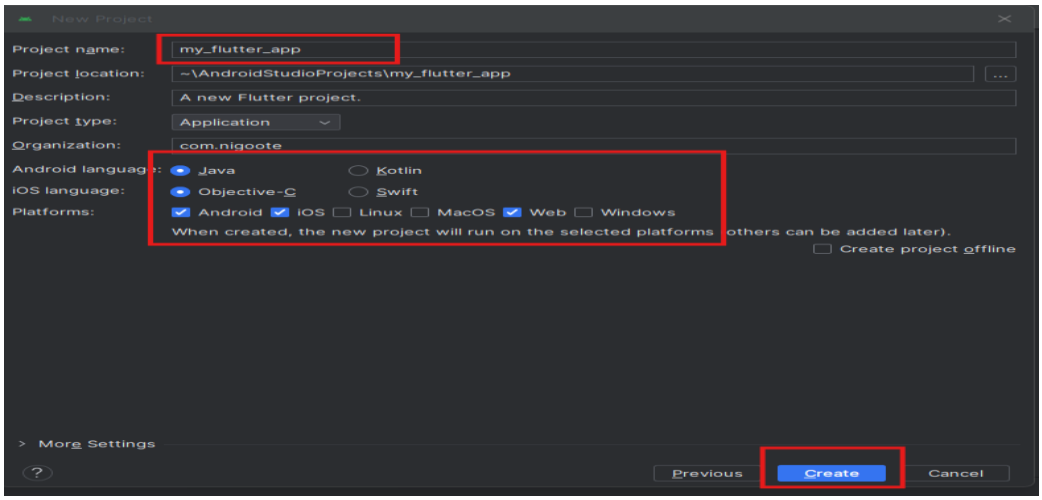
2. Go to **File > New > New Flutter Project**.



3. Choose **Flutter Application** and click **Next**.



4. Set the **Project Name** (e.g., my_flutter_app), choose the Flutter SDK path, and click **Next**.



5. Configure the package name and platform, then click **Finish**.

6. Flutter create project name(create new project using command line Interface)

To create a new Flutter project using the command line interface (CLI), follow these steps:

I. Open your terminal or command prompt.

II. Navigate to the directory where you want to create the new Flutter project.

III. Run the following command:

```
flutter create project_name
```

Replace project_name with the name you want to give your project.

For example:

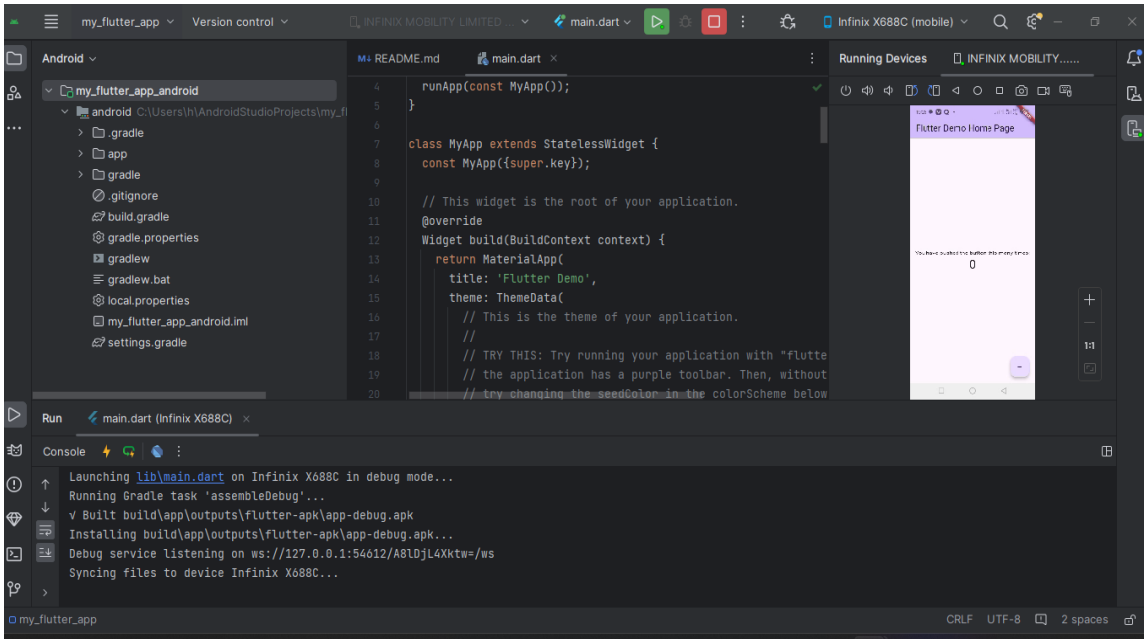
```
flutter create my_motivation_app
```

This command will generate a new Flutter project with the default directory structure, and you can start developing from there. After it's created, you can navigate into your new project folder:

```
cd my_motivation_app
```

5. Create a New Activity Named HomeActivity with a Text Widget

1. Open the lib/main.dart file in your new project.



This is the default program and runs in external Emulator (Mobile Phone)

You can test your Flutter app on mobile devices using two primary methods: via **USB Debugging** (for a physical device) and using an **Emulator**. Here's how to set up both:

1. Testing on a Physical Device via USB Debugging

Step 1: Enable USB Debugging

1. **On Android:**

- Open **Settings** on your Android device.
- Go to **About Phone** and tap on **Build number** seven times to enable Developer Mode.
- Go back to **Settings**>**Developer options** and turn on **USB Debugging**.

2. **On iOS (for macOS users):**

- Connect your device to your Mac.
- Trust your computer on your iPhone.
- Enable **developer mode** if prompted.

Step 2: Connect Your Device

- Connect your Android or iOS device to your computer via USB.

Step 3: Verify Device Connection

- Open the terminal or command prompt and run:

```
flutter devices
```

This will list connected devices. If your device appears in the list, you're ready to run the app.

Step 4: Run the App

- In Android Studio:
- Click the green **Run** button, and select your connected device as the target.
- Via the terminal:
- Run:

```
flutter run
```

- Your app should launch on the connected physical device.

2. Testing on an Emulator

Step 1: Create an Android Emulator or Use iOS Simulator

- **Android Studio (Android Emulator):**

1. Open Android Studio.
2. Go to **AVD Manager** (Android Virtual Device Manager) from the toolbar (click the device icon).

3. Click **Create Virtual Device** and choose a device model.
4. Select a system image (e.g., Android 11 or above), and click **Finish**.
5. Once created, launch the emulator by clicking the green play icon next to the emulator in AVD Manager.

- **Xcode (iOS Simulator - macOS only):**

1. Open Xcode.
2. Go to **Xcode**>**Preferences**>**Components** to install available simulators.
3. Once installed, launch the simulator from **Xcode**>**Open Developer Tool**>**Simulator**.

Step 2: Run the App

- **In Android Studio:**
  - Choose the emulator as the target device, then click the **Run** button.
- **Via Terminal:**
  - Start the emulator using:

```
flutter emulators --launch <emulator_id>
```

  - Run the app:

```
flutter run
```

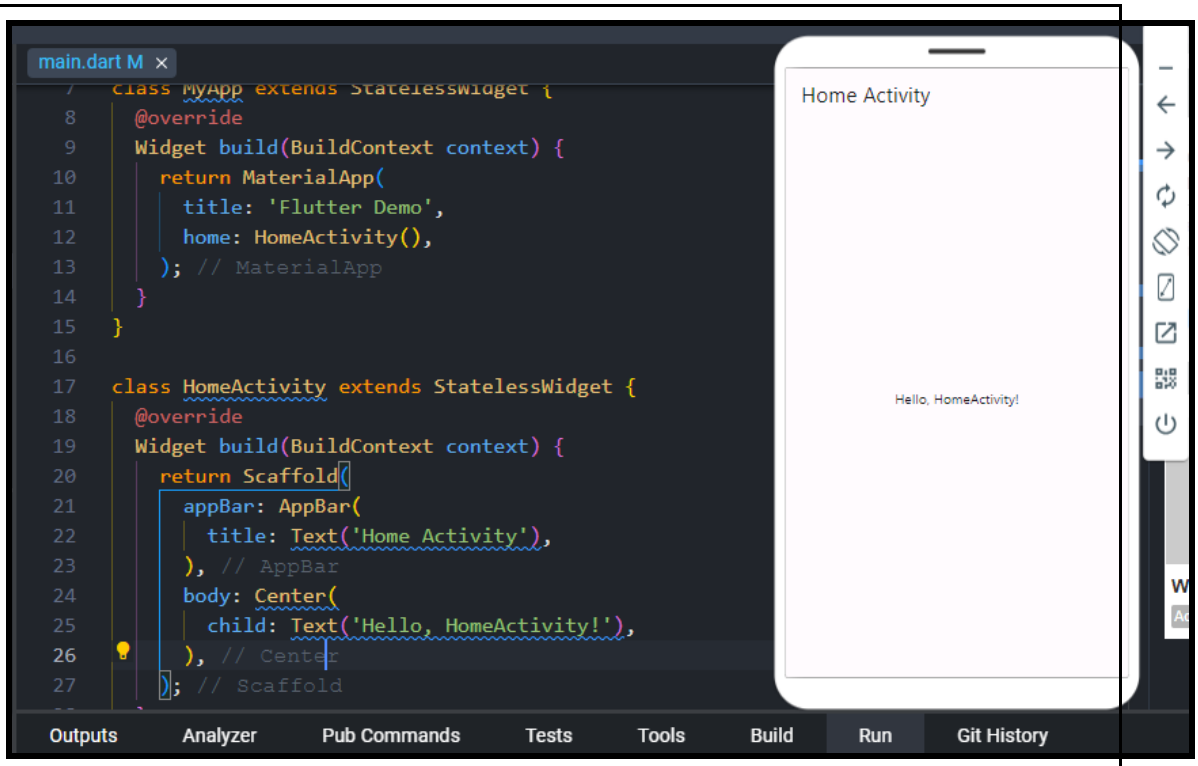
- The app will launch on the emulator or iOS simulator.

These are the two ways to test your Flutter app: on a physical device using USB debugging or on an emulator.

2. Replace the code with the following to create a new HomeActivity:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
```

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



6. Run and Test the Project

1. **In Android Studio:**

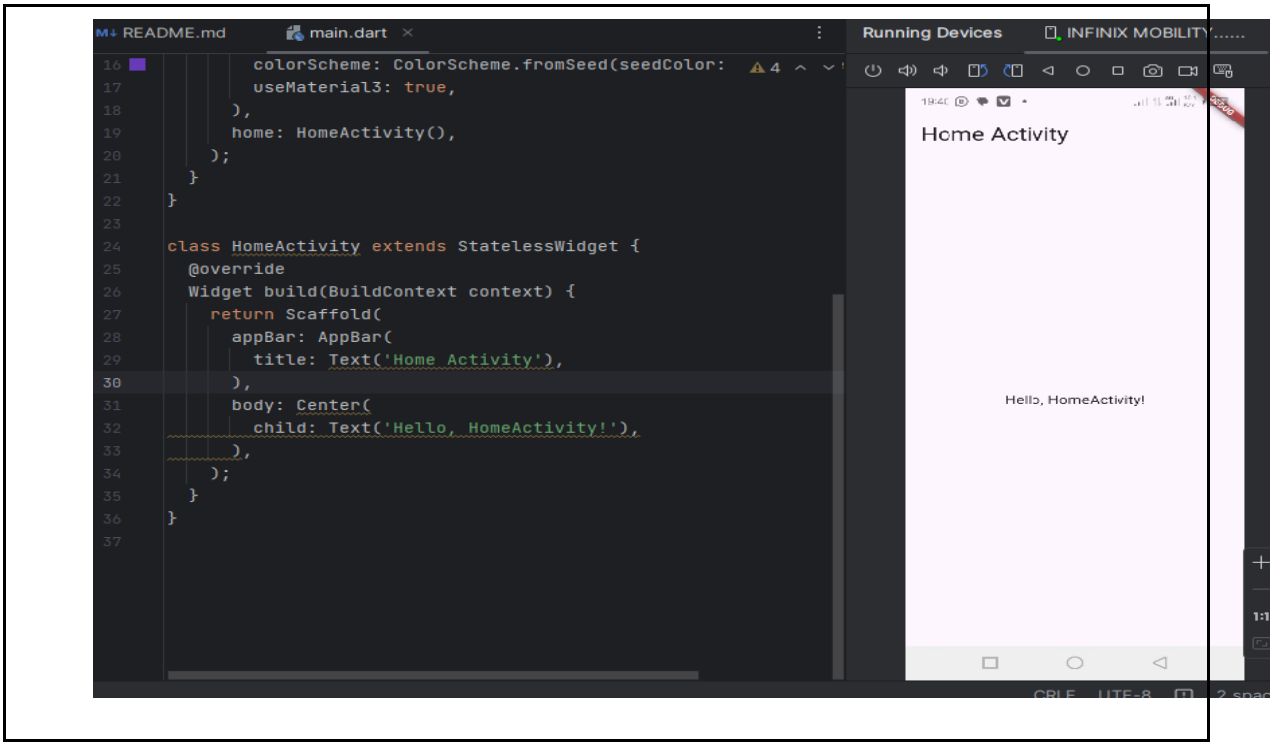
- Select your device (emulator or connected physical device).
- Click the green "Run" button in Android Studio to run the app.

2. **In the terminal:**

- Navigate to the project directory and run:

flutter run

Once you have done this, the app should run, displaying "Hello, HomeActivity!" on the screen.



Points to Remember

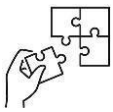
✓ **Description of Flutter framework**

- Flutter is an open-source UI software development kit created by Google. It enables developers to build natively compiled applications for mobile, web, and desktop from a single codebase.
- Flutter's important characteristics include Single Codebase, Rich UI Components, High Performance, Hot Reload, Flexible Layouts, State Management Options.
- The primary purpose of Flutter is to enable developers to create high quality, natively compiled applications for multiple platforms using a single codebase.
- Lifecycle is especially important for **StatefulWidgets**, as it defines the stages of a widget's existence from its creation to its removal.
- Widgets in flutter are broadly categorized into two types: Stateless and Stateful Widgets

✓ **Preparation of flutter environment**

- **While Installing Flutter SDK pass through the following steps:**
  - Download the latest Flutter SDK.
  - Extract the file to your desired location (e.g., C:\src\flutter).
  - Add the Flutter bin directory to your system's PATH.
- **While creating new flutter project pass though the following steps:**

- Open Android Studio
- Go to **File > New > New Flutter Project**.
- Choose **Flutter Application** and click **Next**.
- Set the **Project Name** (e.g., my_flutter_app), choose the Flutter SDK path, and click **Next**.
- Configure the package name and platform, then click **Finish**



Application of learning 2.1.

KUPONI LTD is a software development company located in Kicukiro district, focusing on developing cross-platform mobile applications. They have a project aimed at tracking the fitness of gym members and intend to use Flutter technology for its development. As the lead developer at KUPONI LTD, your role is to help the company prepare the Flutter environment and create a new Flutter project for this initiative.

Indicative content 2.2: Applying flutter's widget system

Duration: 8 hrs

Theoretical Activity 2.2.1 Description of flutter widget's system

Tasks:

1: You are requested to answer the following questions:

I. Differentiate stateful and stateless widget
II. Explain the concept of widget composition in Flutter. How can you combine different widgets to create complex UI layouts?
III. What are the primary differences between the Row and Column widgets?
IV. What is the purpose of the Expanded widget?
V. How can you customize the appearance of text in Flutter, including font, size, color, and alignment

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 2.2.1. In addition, ask questions where necessary.

Key readings 2.2.1:Description of flutter widget's system

✓ **Stateful and Stateless Widgets**

- **Stateless Widgets**: These are immutable and do not store any state. They are built once and their properties are final. Use Stateless widgets when the UI does not change based on user interaction or other events. For example, a simple text label or an icon.
  - **Example**:

```
class MyStatelessWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Text('Hello, Flutter!');
  }
}
```

- **Stateful Widgets**: These can change their state during the app's lifetime. They can rebuild their UI when the internal state changes. Use Stateful widgets for interactive components like forms, buttons, and other elements that require a

dynamic UI.

Example:

```
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(
      children: [
        Text('Counter: $counter'),
        ElevatedButton(
          onPressed: _incrementCounter,
          child: Text('Increment'),
        ),
      ],
    );
  }
}
```

✓ **Widget Tree and Hierarchy**

- **Parent-child Relationships**: In Flutter, widgets are organized in a tree structure, where each widget can have a parent and may have multiple children. The top-level widget is the root of the tree, often a MaterialApp or CupertinoApp.
- **Widget Composition**: You can create complex UIs by combining simple widgets. For instance, you can use Rows and Columns to arrange widgets in a grid-like layout.
- **Row and Column**

Row: Arranges its children in a horizontal manner. It takes up the minimum height required by its children.

Example:

```
Row(
  children: [
    Icon(Icons.favorite),
    Text('Favorite'),
  ],
)
```

Column: Arranges its children in a vertical manner. It takes up the minimum width required by its children.

Example:

```
Column(
  children: [
    Text('Item 1'),
    Text('Item 2'),
  ],
)
```

✓ **Container**

- A versatile widget used for creating rectangular visual elements. It can hold one child and allows you to customize its size, padding, margin, and decoration (like borders and background color).

Example:

```
Container(
  padding: EdgeInsets.all(10),
  margin: EdgeInsets.symmetric(vertical: 10),
  color: Colors.blue,
  child: Text('Hello, Flutter!'),
)
```

✓ **Expanded**

- A widget that expands a child of a Row, Column, or Flex so that it fills the available space. It is often used within Row or Column to distribute space among children.

Example:

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

✓ **Stack**

- A widget that allows you to place children on top of each other. It is useful for creating overlays and complex layouts.

Example:

```
Stack(
  children: [
    Image.asset('background.png'),
    Positioned(
      top: 20,
      left: 20,
      child: Text('Overlay Text'),
    ),
  ],
)
```

✓ **Core Widgets**

- **Text and Styling**: The Text widget is used to display text. You can customize its style with the TextStyle class.

Example:

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

- **Image and Asset**: The Image widget displays images. You can load images from the asset bundle or the network.

Example:

```
Image.asset('assets/images/logo.png')
```

- **Interactive Widgets**: Widgets like ElevatedButton, GestureDetector, and InkWell provide interactivity. Use them to handle user input.

Example:

```
ElevatedButton(
  onPressed: () {
    // Handle button press
  },
  child: Text('Press Me'),
)
```

- **Layout Widgets**: These include Container, Row, Column, Stack, and others that help organize the visual structure of your UI.

Practical Activity 2.2.2: Applying flutter's widget

Task:

1: Read key reading 2.2.1 and ask for clarification where necessary.
2: You are requested to go to the computer lab to create and implement Stateless and Stateful widgets in your Flutter project for managing dynamic and static UI components.
3: Apply safety precautions while working with computer hardware and software.
4: Referring to the steps provided in key readings, create a new Flutter project or use an existing one, and implement both Stateless and Stateful widgets. Use a Stateless Widget for static UI components (e.g., text or images that do not change) and a Stateful Widget for components that respond to user interactions (e.g., buttons that update data).
5: Present your work to the trainer and the whole class, demonstrating how the widgets behave differently, based on their statefulness.

Key readings 2.2.1: Applying flutter's widget

Practical Activity: Flutter UI Development

Step 1: Create a New Flutter Project

1. Open your terminal/command prompt or use **Android Studio**.
2. Run the command to create a new project:
   flutter create FoodieApp
3. Navigate to the project directory:
   cd FoodieApp
4. Open the project in your preferred IDE (Android Studio or VS Code).

Step 2: Add a StatelessWidget

1. In the lib/main.dart file, replace the MyApp class with the following code to create a **Stateless** widget:

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

void main() {
  runApp(MyApp());
}
class MyApp extends StatelessWidget {
  @override
```

```
  Widget build(BuildContext context) {
   return MaterialApp(
    home: HomeScreen(),
   );
  }
}
class HomeScreen extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(
    title: Text('FoodieApp - Stateless Widget'),
   ),
   body: Center(
    child: Text('Welcome to FoodieApp'),
   ),
  );
 }
}
```

Step 3: Add a StatefulWidget

1. In the same lib/main.dart file, create a **Stateful** widget by adding the following code under the HomeScreen class:

```
class FoodCounter extends StatefulWidget {
 @override
 _FoodCounterState createState() => _FoodCounterState();
}

class _FoodCounterState extends State<FoodCounter> {
 int count = 0;

 @override
 Widget build(BuildContext context) {
  return Column(
   mainAxisAlignment: MainAxisAlignment.center,
   children: [
    Text('Items in Cart: $count'),
    ElevatedButton(
     onPressed: () {
      setState(() {
```

```
            count++;
           });
          },
          child: Text('Add to Cart'),
        ),
       ],
     );
   }
}
```

Step 4: Use Row and Column Layouts

1. Update the HomeScreen widget to include **Row** and **Column** layouts, like this:

```
class HomeScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('FoodieApp'),
      ),
      body: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Text('Choose Your Favorite Food'),
          Row(
            mainAxisAlignment: MainAxisAlignment.spaceAround,
            children: [
              Container(
                width: 100,
                height: 100,
                color: Colors.red,
                child: Center(child: Text('Pizza')),
              ),
              Container(
                width: 100,
                height: 100,
                color: Colors.green,
                child: Center(child: Text('Burger')),
              ),
            ],
          ),
```

```
            FoodCounter(),
          ],
        ),
      );
    }
}
```

Step 5: Implement a Container with Expanded

1. Modify the Row layout to wrap each **Container** inside an **Expanded** widget for better responsiveness:

```
Row(
  mainAxisAlignment: MainAxisAlignment.spaceAround,
  children: [
    Expanded(
      child: Container(
        margin: EdgeInsets.all(10),
        height: 100,
        color: Colors.red,
        child: Center(child: Text('Pizza')),
      ),
    ),
    Expanded(
      child: Container(
        margin: EdgeInsets.all(10),
        height: 100,
        color: Colors.green,
        child: Center(child: Text('Burger')),
      ),
    ),
  ],
),
```

Step 6: Include Text, Image, and Interactive Buttons

1. Add an **Image** widget for displaying food images:

```
Image.network(
  'https://example.com/food_image.png',
  height: 150,
  width: 150,
),
```

2. Add an interactive button to navigate to a detailed screen:

```
ElevatedButton(
  onPressed: () {
    // Perform action here
  },
  child: Text('Order Now'),
),
```

Step 7: Visualize the Widget Tree and Explain Parent-Child Relationships

1. Use the **Flutter Inspector** in your IDE to view the widget tree. This will show how the **Column**, **Row**, **Container**, and other widgets are nested, revealing the **parent-child relationships**.
2. Explain how:
   - The **Column** is the parent of the **Text**, **Row**, and **FoodCounter** widgets.
   - The **Row** contains **Expanded** widgets as children, which in turn contain **Container** widgets.

Step 8: Run and Test the App

1. Run the app in your emulator or on a real device:

```
flutter run
```

2. Test the interaction by clicking the "Add to Cart" button to see the **StatefulWidget** in action and view the food items displayed with **Stateless** widgets.

Expected Output:

- A main screen that displays food categories in a **Column** layout.
- A **Row** showing food options with proper spacing using **Expanded** containers.
- **Text** and **Image** widgets displaying food details.
- A **Stateful** button that adds items to the cart when clicked, demonstrating how state changes.

Note that: This practical activity covers all the key concepts like **Stateless/Stateful widgets**, **Row/Column layouts**, **Containers**, **Text**, **Images**, and **Interactive buttons**, while focusing on parent-child relationships and the widget tree.

Points to Remember

- **Stateless Widget**: Used for widgets that don't need to track any changeable state. Once created, they do not change unless explicitly rebuilt.
- **Parent-Child Relationships**: In Flutter, the UI is a hierarchy of widgets where parent widgets contain child widgets. The parent controls layout, position, and behavior.
- **Row and Column**: These are layout widgets used to arrange children in horizontal (Row) or vertical (Column) directions.
- **Expanded**: Used within Row or Column to allow children to take up available space proportionally.
- **Stack**: A layout widget that positions its children on top of each other, useful for overlay designs.
- **Image and Asset**: The Image widget displays images from assets, network, or files, while managing sizes and alignment.
- **Interactive Widgets (Buttons, Gesture**): Widgets like ElevatedButton, FlatButton, and GestureDetector allow user interaction through taps and gestures.
- **Layout Widgets**: These widgets help define the structure and arrangement of child widgets on the screen, such as Container, Row, Column, Stack, and Expanded

While Applying flutter's widget, pass through the following steps:

- Create a new Flutter project using the command flutter create project_name.
- Open the project in your IDE (e.g., Android Studio or VS Code).
- Add a Stateless widget for static UI components.
- Add a Stateful widget for interactive UI components.
- Use Row and Column widgets to arrange child widgets horizontally and vertically, respectively.
- Implement a Container widget to wrap and style other widgets.
- Use the Expanded widget to allow children to fill available space.
- Include a Text widget to display text, an Image widget for displaying images, and interactive buttons for user interaction.
- Visualize the widget tree to understand parent-child relationships in your layout

Application of learning 2.2.

Foodie Delights, a food delivery service in Kigali, has hired you as a Flutter developer to create FoodieApp, a mobile application for browsing and ordering food from various restaurants. As mobile developer, you are requested to set up the development environment, create a new Flutter project and build the UI with Statefull and Stateless widgets. You'll use a Column layout for food categories, a Row layout for restaurant listings, and Container widgets for styling. Text and Image widgets will display food details, while interactive buttons will allow users to add items to their cart. All necessary resources will be provided to ensure smooth development.

Indicative content 2.3: Implementation of State Management

Duration: 8 hrs

Theoretical Activity 2.3.1: Description of state management

Tasks:

1: You are requested to answer the following questions:

i. What is the GetX package, and how does it improve state management in Flutter applications?
ii. Explain how the Provider package is used for managing state in a Flutter app. How does it inject dependencies into the widget tree?
iii. Compare the use of GetX and Provider for state management. What are the advantages of each?
iv. What is the Redux pattern, and how does it manage state in Flutter applications?
v. What does the setState() method do in Flutter, and when is it used?
vi. Explain how setState() affects the lifecycle of a widget and how it triggers UI updates
vii. What is Riverpod, and how does it differ from Provider in terms of state management?
viii. What is the Navigator widget in Flutter, and how is it used to manage routes between screens?

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 2.3.1.

Key readings 2.3.1.: Description of state management

In Flutter, **packages** are pre-built sets of code that provide various functionalities to speed up development. Flutter's ecosystem offers many useful packages that developers can integrate into their projects. We'll look at two common state management packages: **GetX** and **Provider**.

a. GetX Package (Zero to Expert)

Beginner Level:

- **What is GetX?**
  - GetX is an open-source Flutter package that simplifies state management, route management, and dependency injection. It helps keep the

app reactive and lightweight.

- **Installing GetX**:
  - Add GetX to your pubspec.yaml file:

```
dependencies:
  get: ^4.6.5  # latest version
```

Import GetX:

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

- **Using GetX for State Management**:

```
class CounterController extends GetxController {
  var count = 0.obs;  // Observable variable
  void increment() => count++;  // Method to increment count
}

// In your UI:
final CounterController controller = Get.put(CounterController());

Obx(() => Text('${controller.count}'));  // Reactive widget
```

Intermediate Level:

- **Routing with GetX**:

Define routes using GetMaterialApp:

```
GetMaterialApp(
  initialRoute: '/',
  getPages: [
    GetPage(name: '/', page: () => HomeScreen()),
    GetPage(name: '/details', page: () => DetailScreen()),
  ],
);
```

  - Navigate between screens:

```
Get.to(DetailScreen());
```

Expert Level:

- **Dependency Injection with GetX**:
  - Inject dependencies into your classes effortlessly:

```
class MyService extends GetxService {
```

```
  Future<MyService> init() async {
    print("Service initialized");
    return this;
  }
}
```

b. Provider Package (Zero to Expert)

Beginner Level:

- **What is Provider?**
  - Provider is another package used for state management. It wraps the InheritedWidget and simplifies state management by efficiently notifying the UI about state changes.
- **Installing Provider**:
  - Add Provider to your pubspec.yaml:

```
dependencies:
  provider: ^6.0.0
```

- **Basic Usage**:

```
class Counter with ChangeNotifier {
  int _count = 0;
  int get count => _count;

  void increment() {
    _count++;
    notifyListeners();  // Notify listeners when state changes
  }
}

ChangeNotifierProvider(
  create: (context) => Counter(),
  child: MyApp(),
);

// In the widget tree:
final counter = Provider.of<Counter>(context);
Text('${counter.count}');
```

Intermediate Level:

- **Selector and Consumer**:
  - Consumer rebuilds specific parts of the widget tree:

```
Consumer<Counter>(
  builder: (context, counter, child) => Text('${counter.count}'),
);
```

Expert Level:

- **MultiProvider**: Providing multiple models:

```
MultiProvider(
  providers: [
    ChangeNotifierProvider(create: (_) => Counter()),
    ChangeNotifierProvider(create: (_) => AnotherModel()),
  ],
  child: MyApp(),
);
```

2. Using Patterns

Flutter offers multiple patterns for handling the architecture of your application, focusing on how to manage state, business logic, and communication between components.

a. Redux Pattern (Zero to Expert)

Beginner Level:

- **What is Redux?**
  - Redux is a predictable state container for Dart and Flutter applications. It centralizes state and uses actions and reducers to manage how the state changes.
- **Core Redux Concepts**:
  - **State**: The entire app's state is stored in one object.
  - **Actions**: Describes changes to state.
  - **Reducers**: Functions that update the state based on actions.

Intermediate Level:

- **Setting Up Redux**:
  - Add the Redux package:

```
dependencies:
  flutter_redux: ^0.8.2
```

  - Create actions:

```
class IncrementAction {}
```

  - Create a reducer:

```
int counterReducer(int state, dynamic action) {
  if (action is IncrementAction) {
    return state + 1;
  }
  return state;
}
```

- Create a store:

```
final store = Store<int>(counterReducer, initialState: 0);
```

Expert Level:

- **Middleware**: Implement custom logic like logging or API requests using middleware in Redux.

```
void loggingMiddleware(Store<int> store, action, NextDispatcher next) {
  print('action: $action');
  next(action);
}
```

b. BLoC Pattern (Zero to Expert)

Beginner Level:

- **What is BLoC?**
  - BLoC (Business Logic Component) separates business logic from UI components. It relies on Streams to handle communication between components.

Intermediate Level:

- **Basic BLoC Setup**:
  - Define a Stream:

```
class CounterBloc {
  final _counterStreamController = StreamController<int>();
  Stream<int> get counter => _counterStreamController.stream;

  void increment() {
    _counterStreamController.sink.add(_count++);
  }

  void dispose() {
    _counterStreamController.close();
  }
```

}

Expert Level:

- **Advanced BLoC with Events and States**:
  - Instead of working with direct streams, events and states provide better management of complex logic.

3. Using setState() Method

a. Beginner Level:

- **What is setState()?**
  - In Flutter, the setState() method tells the framework that the state of the widget has changed, and the UI needs to be rebuilt.
- **Example Usage**:

```
class MyHomePageState extends State<MyHomePage> {
 int _counter = 0;

 void _incrementCounter() {
  setState(() {  // Update state
   _counter++;
  });
 }
}
```

b. Intermediate Level:

- **Performance Considerations**:
  - Avoid using setState() too frequently, especially with heavy widgets, to prevent performance issues.

c. Expert Level:

- **Optimizing with Subwidgets**:
  - Split large widgets into smaller subwidgets and use setState() only where needed.

4. Using Riverpod Solution (Zero to Expert)

Beginner Level:

- **What is Riverpod?**
  - Riverpod is a modern, safer, and more scalable alternative to Provider for state management. It provides better compile-time checks and performance.

Intermediate Level:

- **Basic Setup**:
  - Define a Provider:

```
final counterProvider = StateProvider((ref) => 0);
```

- Use in a widget:

```
Consumer(
  builder: (context, watch, _) {
    final count = watch(counterProvider).state;
    return Text('$count');
  },
);
```

Expert Level:

- **Advanced Riverpod**:
  - Use FutureProvider for handling async logic:

```
final dataProvider = FutureProvider<List<String>>((ref) async {
  return fetchData();
});
```

5. Using Navigation and Routing (Zero to Expert)

a. Navigator and Route

Beginner Level:

- **Navigator**:
  - Navigator helps you move between different screens in your app.
  - Example:

```
Navigator.push(context, MaterialPageRoute(builder: (context) => SecondScreen()));
```

Intermediate Level:

- **Named Routes**:
  - You can set up named routes for easier navigation.

```
Navigator.pushNamed(context, '/second');
```

Expert Level:

- **Custom Transitions**: Use custom transitions with PageRouteBuilder.

b. BottomNavigationBar & TabBar

Beginner Level:

- **BottomNavigationBar**:
  - Used to navigate between different sections of an app.
- **TabBar**:

- Allows for navigation using tabs.

```
DefaultTabController(
  length: 3,
  child: Scaffold(
    appBar: AppBar(
      bottom: TabBar(
        tabs: [
          Tab(icon: Icon(Icons.home)),
          Tab(icon: Icon(Icons.settings)),
        ],
      ),
    ),
    body: TabBarView(
      children: [HomeScreen(), SettingsScreen()],
    ),
  ),
);
```

Practical Activity 2.3.2: Implementing state management

Task:

1:As a mobile app developer, you are tasked with creating a Flutter counter app using the **GetX** package for state management. When a button is pressed, the app should increase the count displayed on the screen.

2:Before starting the task, make sure you understand the following:

- **Install the GetX package** in your Flutter project.
- **Create a CounterController** to manage the counter state using GetX.
- **Use the GetX widget** to display the counter value and update it reactively when the button is pressed.

3: Have you installed the GetX package in your project? If not, please add it to your pubspec.yaml file and run flutter pub get.

4: Can you create a CounterController that extends GetxController and manage an observable integer?

5: Have you used the GetX or Obx widget in your UI to display the counter value?

6: Does the counter update reactively when you press the button? If not, review the state management setup.
7: Have you read key reading 2.3.1 in the manual on GetX state management? If not, please do so to strengthen your understanding.

Key readings 2.2.1: Implementing state management

Step 1: Install GetX Package

First, ensure you have the GetX package installed in your Flutter project. Open your pubspec.yaml file and add GetX to the dependencies:

```
dependencies:
  flutter:
    sdk: flutter
  get: ^4.6.5  # Add this line
```

After adding this line, run the following command in your terminal to install the package:

```
flutter pub get
```

Step 2: Create a CounterController

Create a new Dart file for your controller, for example, counter_controller.dart. This controller will extend GetxController and manage an observable integer for the counter.

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

class CounterController extends GetxController {
  // Observable integer variable for the counter
  var count = 0.obs;

  // Method to increment the counter
```

```
  void increment() {
    count++;
  }
}
```

Step 3: Implement the UI

Now, create the main UI of your app. In your main.dart file, set up the application to use GetX for state management and create the UI to display the counter and button.

```
import 'package:flutter/material.dart';
import 'package:get/get.dart';
import 'counter_controller.dart'; // Import the CounterController

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return GetMaterialApp(
      title: 'GetX Counter App',
      home: CounterScreen(),
    );
  }
}

class CounterScreen extends StatelessWidget {
```

```
// Instantiate the CounterController using GetX
final CounterController controller = Get.put(CounterController());


@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text('GetX Counter'),
    ),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: <Widget>[
          // Use Obx to make the Text widget reactive
          Obx(() => Text(
                'Count: ${controller.count}',
                style: TextStyle(fontSize: 32),
              )),
          SizedBox(height: 20),
          ElevatedButton(
            onPressed: () {
              controller.increment();  // Call increment method on button press
            },
            child: Text('Increment'),
          ),
        ],
      ),
```

```
      ),
    );
  }
}
```

Step 4: Run the Application

Now that you have set up everything, run your Flutter application:

```
flutter run
```

How It Works

1. **CounterController**:
   - The CounterController class manages the counter state. The count variable is an observable, meaning any UI that listens to it will reactively update when it changes.
2. **GetX Widget**:
   - In the CounterScreen class, Obx is used to wrap the Text widget. This ensures that the text is updated whenever controller.count changes.
3. **Incrementing the Counter**:
   - When the button is pressed, the increment method is called, which increments the count. Since count is observable, the Text widget automatically updates to reflect the new count.

Final Steps and Troubleshooting

- **Reactive Updates**: Ensure that the Obx widget wraps the widget you want to update reactively. If you notice that the counter does not update, check that you are using .obs for the observable variable and that you have properly set up your controller.
- **Additional Features**: You can enhance the app by adding a reset button or displaying a decrement button, following the same pattern used for incrementing.

Conclusion

This example provides a simple yet effective way to understand how to use the GetX package in a Flutter application for state management. You can build upon this foundation to create more complex applications using GetX. If you have any questions or need further assistance, feel free to ask!

Points to Remember

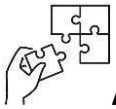
Using Packages

- **GetX Package**:
  GetX is a lightweight state management, navigation, and dependency injection library in Flutter. It offers a reactive approach with minimal boilerplate, making it ideal for handling large-scale applications with better performance.
- **Provider Package**:
  Provider is a widely-used state management package in Flutter. It uses the InheritedWidget under the hood to manage and inject dependencies into the widget tree, allowing widgets to react to state changes.
- **Redux Pattern**:
  Redux is a predictable state container where the entire application state is stored in a single object, known as the store. State updates are handled through actions and reducers, ensuring that state changes are predictable and centralized.
- **setState() Method**:
  The setState() method is used to rebuild the widget tree when there's a change in the state of a StatefulWidget. It's simple to use but can become inefficient in complex applications, especially when frequent updates are needed across different parts of the app.
- **Riverpod**:
  Riverpod is a state management solution in Flutter that is similar to Provider but offers better modularity, testability, and flexibility. It helps in managing state outside the widget tree and allows more control over how state is created, updated, and disposed of.

While implementing state management, pass through the following steps:

- **Using GetX for State Management**
  - ✓ Installation of GetX follows these steps:
  - ✓ Add GetX dependency to pubspec.yaml.
  - ✓ Run flutter pub get to install the package.
  - ✓ Create a Controller class using GetxController.
  - ✓ Use the GetX widget to reactively update the UI.
  - ✓ Call Get.put() to initialize the controller in your widget tree.
- **Using Patterns (Redux and BLoC)**
  - ✓ For Redux pattern:
  - ✓ Install the redux package.
  - ✓ Create a store to hold the global app state.
  - ✓ Define actions and reducers to update the state.

- ✓ Dispatch actions to update the state using the store.dispatch() method.



Application of learning 2.3.

Nigoote Ltd, a leading software company based in Kigali, specializes in building high-quality mobile and web applications. Due to their growing client base. As mobile developer, you are requested to implement: GetX package, provider package, Redux pattern , Business Logic Component (BLoC) pattern, setState() method, Riverpod, Route,BottomNavigationBar, Navigator, TabBar and TabBarView.

Indicative content 2.4: Using Pre-designed Widgets

Duration: 7 hrs

Theoretical Activity 2.4.1: Description of pre-defined widgets

Tasks:

1: You are requested to answer the following questions related to the description of authorization:

I. Which of the following options is a key characteristic of Cupertino Widgets in Flutter?

A) They follow Material Design guidelines.

B) They provide an iOS-style UI experience.

C) They are only available on Android devices.

D) They do not support animations.

II. What command do you use to add a third-party package to your Flutter project and manage dependencies?

A) flutter pub add

B) flutter install package

C) flutter get package

D) flutter pub get

III. Which widget is used to implement a Floating Action Button in Flutter, and how would you display an icon within it?

A) FloatingActionButton(child: Icon(Icons.add))

B) IconButton(FloatingActionButton: Icons.add)

C) Icon(Icons.fab)

D) FloatingActionButton(Icons.add)

IV. Can you explain the differences between Material Design Widgets and Flutter Icons, particularly in their usage and appearance?

V. What is the significance of using third-party packages in a Flutter application, and can you name a popular package you've used?

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 2.4.1. In addition, ask questionswhere necessary.

Key readings 2.4.1: Description of pre-defined widgets

1. Material Design Widgets

Overview:

Material Design is a design system developed by Google, focusing on providing a modern and consistent user experience across all Android devices. Flutter's Material Design widgets make it easy to create apps that follow these principles.

Key Widgets:

- **Scaffold**: The base structure of a screen in a Material Design app. It provides functionality like app bars, drawers, and bottom navigation bars.
- **AppBar**: A customizable toolbar usually at the top of the screen.
- **FloatingActionButton**: A circular button that typically triggers a primary action on a screen.
- **ListTile**: A highly customizable list item widget, commonly used to represent a single row in a list.

Example:

```
Scaffold(
  appBar: AppBar(
    title: Text("Material Design Example"),
  ),
  body: Center(
    child: Text("Hello, Material Design!"),
  ),
  floatingActionButton: FloatingActionButton(
    onPressed: () {
      // Action here
    },
    child: Icon(Icons.add),
  ),
);
```

Key Points:

- **Consistency**: Using Material Design widgets ensures consistency across Android apps.
- **Ease of Use**: Widgets like Scaffold, AppBar, and FloatingActionButton come pre-built with common functionalities.
- **Customization**: Material widgets are highly customizable, making it easy to change colors, shapes, and behaviors.

2. Cupertino Widgets

Overview:

Cupertino widgets bring iOS-style design elements to Flutter apps. They follow Apple's Human Interface Guidelines to create a native look and feel for iOS apps.

Key Widgets:

- **CupertinoButton**: A button that follows iOS styles.
- **CupertinoNavigationBar**: A navigation bar that sits at the top of the screen, similar to the AppBar but styled for iOS.
- **CupertinoPicker**: A customizable picker (used for selecting items like dates, times, or lists).
- **CupertinoSwitch**: A toggle switch that conforms to iOS design.

Example:

```
CupertinoPageScaffold(
 navigationBar: CupertinoNavigationBar(
  middle: Text("Cupertino Example"),
 ),
 child: Center(
  child: CupertinoButton(
   onPressed: () {
    // Action here
   },
   child: Text("Press me"),
  ),
 ),
);
```

Key Points:

- **iOS Styling**: Use Cupertino widgets to create apps that look native on iOS devices.
- **Cross-Platform Strategy**: You can mix Cupertino and Material widgets to support both Android and iOS platforms.
- **Smooth Animations**: Cupertino widgets come with smooth, native-looking animations for interactions and transitions.

3. Flutter Icons

Overview:

Icons play a crucial role in enhancing the UI of an app. Flutter provides a rich set of pre-designed icons for both Material and Cupertino designs.

Material Icons:

The Material icon set is extensive and covers a wide variety of common icons.

Icon(Icons.home) // Home icon from Material Icons

Cupertino Icons:

Cupertino Icons are styled according to iOS standards.

```
Icon(CupertinoIcons.heart)  // Heart icon from Cupertino Icons
```

Customizing Icons:

Icons in Flutter can be resized and colored to match the app's theme.

```
Icon(
  Icons.star,
  size: 30.0,
  color: Colors.yellow,
);
```

Key Points:

- **Consistency in Iconography**: Using Flutter Icons ensures you're adhering to platform guidelines for both Android and iOS.
- **Customization**: Icons are highly flexible, and you can easily adjust their size, color, and alignment.
- **Cross-Platform**: Material and Cupertino icons allow for a consistent experience across both platforms, with their native designs.

4. Third-Party Packages

Overview:

Flutter's ecosystem includes a wide range of third-party packages that can add advanced functionalities without the need to reinvent the wheel. These packages are available on pub.dev, Flutter's official package repository.

Popular Third-Party Packages:

- **http**: For making HTTP requests.
- **provider**: A popular state management solution.
- **lottie**: For adding complex animations in your Flutter apps.
- **flutter_slidable**: For creating swipeable list items with actions.

Adding a Package:

To use a third-party package, add it to your pubspec.yaml file:

```
dependencies:
  http: ^0.13.3
```

Then, run:

```
flutter pub get
```

Example: Using the http package:

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

  if (response.statusCode == 200) {
    print('Data: ${response.body}');
  } else {
    print('Failed to load data');
  }
}
```

Key Points:

- **Community Support**: Third-party packages extend the functionality of Flutter apps, leveraging community-built solutions.
- **Ease of Integration**: Packages are easy to integrate and follow a standardized process via pubspec.yaml.
- **Diverse Functionality**: There are packages for nearly every kind of functionality, from HTTP requests to animations and beyond.

5. Cross-Platform Development

Overview:

One of Flutter's greatest strengths is its ability to create cross-platform apps using a single codebase. Material widgets and Cupertino widgets can be used together to ensure your app looks and feels native on both Android and iOS.

Using Platform-Specific Widgets:

You can use platform-specific checks to determine whether to show a Material or Cupertino widget:

```
import 'dart:io';
Widget build(BuildContext context) {
  if (Platform.isIOS) {
    return CupertinoButton(
      onPressed: () {},
      child: Text("iOS Button"),
    );
  } else {
    return ElevatedButton(
      onPressed: () {},
      child: Text("Android Button"),
    );
  }}
```

Practical Activity 2.4.2: Performing pre-designed widgets

Task:

1: Using **pre-designed widgets** in Flutter, focusing on **Material Design Widgets**, **Cupertino Widgets**, **Flutter Icons**, and some **third-party packages**. The goal is to create a simple cross-platform **To-Do List App** where users can add tasks, view them in a list, and mark them as completed.

Activity: To-Do List App Using Pre-Designed Widgets

Objective:

Build a To-Do List app that uses **Material Design Widgets** for Android, **Cupertino Widgets** for iOS, and includes icons and a third-party package for enhanced functionality.

Requirements:

1. **Material Design Widgets** for Android UI.
2. **Cupertino Widgets** for iOS UI.
3. **Flutter Icons** to add visual elements.
4. **Third-Party Package** to implement swipe-to-delete functionality.
5. **Cross-Platform** design considerations (use platform checks).

Key readings 2.4.2: Performing Pre-designed widgets

Step-by-Step Guide:

1. Set Up the Flutter Project:

Create a new Flutter project using:

flutter create todo_app

cd todo_app

2. Add Third-Party Package:

In the pubspec.yaml file, add the flutter_slidable package for swipe-to-delete functionality:

```
dependencies:
  flutter:
    sdk: flutter
  flutter_slidable: ^0.6.0
```

Run flutter pub get to install the package.

3. Create a Simple To-Do List UI:

In the lib folder, create a home.dart file, and build the main UI.

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

```
import 'package:flutter/cupertino.dart';
import 'package:flutter_slidable/flutter_slidable.dart';
import 'dart:io';
void main() {
  runApp(TodoApp());
}
class TodoApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Platform.isIOS
        ? CupertinoApp(
            home: HomeScreen(),
            theme: CupertinoThemeData(primaryColor: CupertinoColors.activeBlue),
          )
        : MaterialApp(
            home: HomeScreen(),
            theme: ThemeData(primarySwatch: Colors.blue),
          );
  }}
class HomeScreen extends StatefulWidget {
  @override
  _HomeScreenState createState() => _HomeScreenState();
}
class _HomeScreenState extends State<HomeScreen> {
  final List<String> _tasks = [];
  final TextEditingController _controller = TextEditingController();
  void _addTask(String task) {
    setState(() {
      _tasks.add(task);
    });
    _controller.clear();
  }
  void _deleteTask(int index) {
    setState(() {
      _tasks.removeAt(index);
    });
  }
  @override
  Widget build(BuildContext context) {
    return Platform.isIOS
```

```
      ? CupertinoPageScaffold(
          navigationBar: CupertinoNavigationBar(
            middle: Text("To-Do List"),
          ),
          child: _buildTaskList(),
        )
      : Scaffold(
          appBar: AppBar(
            title: Text("To-Do List"),
          ),
          body: _buildTaskList(),
          floatingActionButton: FloatingActionButton(
            onPressed: () => _showAddTaskDialog(context),
            child: Icon(Icons.add),
          ),
        );
  }
  // Task List UI
  Widget _buildTaskList() {
    return ListView.builder(
      itemCount: _tasks.length,
      itemBuilder: (context, index) {
        return Slidable(
          actionPane: SlidableDrawerActionPane(),
          secondaryActions: [
            IconSlideAction(
              caption: 'Delete',
              color: Colors.red,
              icon: Icons.delete,
              onTap: () => _deleteTask(index),
            ),
          ],
          child: ListTile(
            title: Text(_tasks[index]),
          ),
        );
      },
    );
  }
```

```
// Add Task Dialog
void _showAddTaskDialog(BuildContext context) {
  Platform.isIOS
      ? showCupertinoDialog(
          context: context,
          builder: (context) {
            return CupertinoAlertDialog(
              title: Text('Add Task'),
              content: CupertinoTextField(controller: _controller),
              actions: [
                CupertinoDialogAction(
                  child: Text('Cancel'),
                  onPressed: () => Navigator.of(context).pop(),
                ),
                CupertinoDialogAction(
                  child: Text('Add'),
                  onPressed: () {
                    if (_controller.text.isNotEmpty) {
                      _addTask(_controller.text);
                      Navigator.of(context).pop();
                    }
                  },
                ),
              ],
            );
          },
        )
      : showDialog(
          context: context,
          builder: (context) {
            return AlertDialog(
              title: Text('Add Task'),
              content: TextField(
                controller: _controller,
                decoration: InputDecoration(hintText: 'Enter task name'),
              ),
              actions: [
                TextButton(
                  onPressed: () => Navigator.of(context).pop(),
                  child: Text('Cancel'),
```

```
              ),
              TextButton(
                onPressed: () {
                  if (_controller.text.isNotEmpty) {
                    _addTask(_controller.text);
                    Navigator.of(context).pop();
                  }
                },
                child: Text('Add'),
              ),
            ],
          );
        },
      );
  }
}
```

4. Explanation:

1. **Platform Checks**:
   - The Platform.isIOS check ensures that the app uses Cupertino Widgets for iOS and Material Widgets for Android. This is critical for maintaining a native look and feel across both platforms.
2. **Task List UI**:
   - The list of tasks is displayed using a ListView.builder, which dynamically generates the list items.
   - Each list item is wrapped in a Slidable widget from the flutter_slidable package, allowing users to swipe the item to reveal a delete option.
3. **Adding Tasks**:
   - The user can add tasks using a floating action button (FAB) on Android or a button in the navigation bar on iOS.
   - A platform-specific dialog box (CupertinoAlertDialog for iOS, AlertDialog for Android) is used to input the new task.
4. **Deleting Tasks**:
   - When a user swipes a task item, they can delete it using the IconSlideAction with a delete icon.

5. Run the App:

Run the app using the following command:

```
flutter run
```

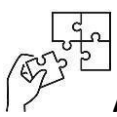
Test the app on both Android and iOS simulators or devices to see the platform-specific widgets in action.

Points to Remember

- **Material Design Widgets**: These widgets follow Google's Material Design guidelines and provide a consistent, modern look and feel for Android apps. Common examples include Scaffold, AppBar, FloatingActionButton, and Card.
- **Cupertino Widgets**: Designed to mimic the native iOS design system, these widgets are essential for building iOS-style apps in Flutter. Widgets like CupertinoNavigationBar, CupertinoButton, and CupertinoTabBar provide iOS-specific look and behavior.
- **Cross-Platform Design**: You can mix Material and Cupertino widgets in a single Flutter project to deliver a native experience on both Android and iOS platforms, ensuring seamless design across devices.
- **Flutter Icons**: The Icon widget allows you to include a variety of pre-designed icons in your app. You can use Icons for Material icons or CupertinoIcons for iOS-style icons, depending on the platform design.
- **Customization of Pre-Designed Widgets**: Both Material and Cupertino widgets are highly customizable. You can easily change colors, shapes, and behaviors to match the desired app theme without needing to build widgets from scratch.
- **Third-Party Packages for Enhanced Widgets**: Many third-party packages are available on Pub.dev, offering pre-built widgets that enhance your app's functionality. Examples include packages like flutter_slidable for swipeable list items and lottie for complex animations.
- **Dependency Management**: When using third-party packages, always manage your pubspec.yaml file carefully and run flutter pub get to install new packages. Ensure the package is compatible with the Flutter version you are using and review the documentation for proper implementation.

While performing pre-designed widgets, pass through the following steps:

- Perform material Design vs. Cupertino
    - ✓ Use Material Design Widgets
- Perform platform-Specific Checks
- Select Icons for Visual Elements.
- Perform Third-Party Packages Extend Functionality
- Perform Cross-Platform Testing



Application of learning 2.4

TechNova Ltd is a mobile app development company located in Kigali, Rwanda, specialized in creating innovative and user-friendly applications for clients in various industries. As a mobile developer you are requested to perform material Design vs.

Cupertino Using Material Design Widgets, platform-Specific Checks, select Icons for Visual Elements, third-Party Packages Extend Functionality, apply Cross-Platform Testing.

Learning outcome 2 end assessment

Theoretical assessment

I. Fill in the empty space by using the correct world listed in blacket(widget, build, natively compiled, Navigator, flutter doctor, core widget)

A. Flutter is an open-source UI software development kit (SDK) created by Google. Its primary purpose is to allow developers to build ……..applications for mobile, web, and desktop from a single codebase.

B. A widget in Flutter is the basic building block of the Flutter UI. Everything in Flutter is a……., including layout structures, elements, and animations.

C. The lifecycle of a Stateful widget includes methods such as createState, initState, and……..

D. To check if everything is properly installed after setting up the Flutter SDK, you run the command:…….

E. The ……..widget is used to navigate between different screens in Flutter.

II. Circle the letter corresponding to the correct answer:

1. Which of the following are features of Flutter?
    a) Hot Reload
    b) Cross-Platform Development
    c) Native Performance
    d) All the above

2. What type of widget would you use if the UI needs to change dynamically based on user interaction?
    a) Stateless Widget
    b) Stateful Widget
    c) Functional Widget
    d) Custom Widget

3. Which state management package is commonly used with Flutter?
    a) Redux
    b) Provider
    c) GetX
    d) All of the above

4. Which widget would you use to display text with specific styling in Flutter?
    A) Text B) RichText C) TextFormField D) All of the above

5. What is the primary difference between Material Design and Cupertino widgets?
   a) Material Design is for iOS, Cupertino is for Android
   b) Material Design follows Android guidelines, Cupertino follows iOS guidelines
   c) They are the same
   d) Material Design is for web apps, Cupertino is for mobile apps

III. Match the following terms to their description and write letters in empty space of answers:

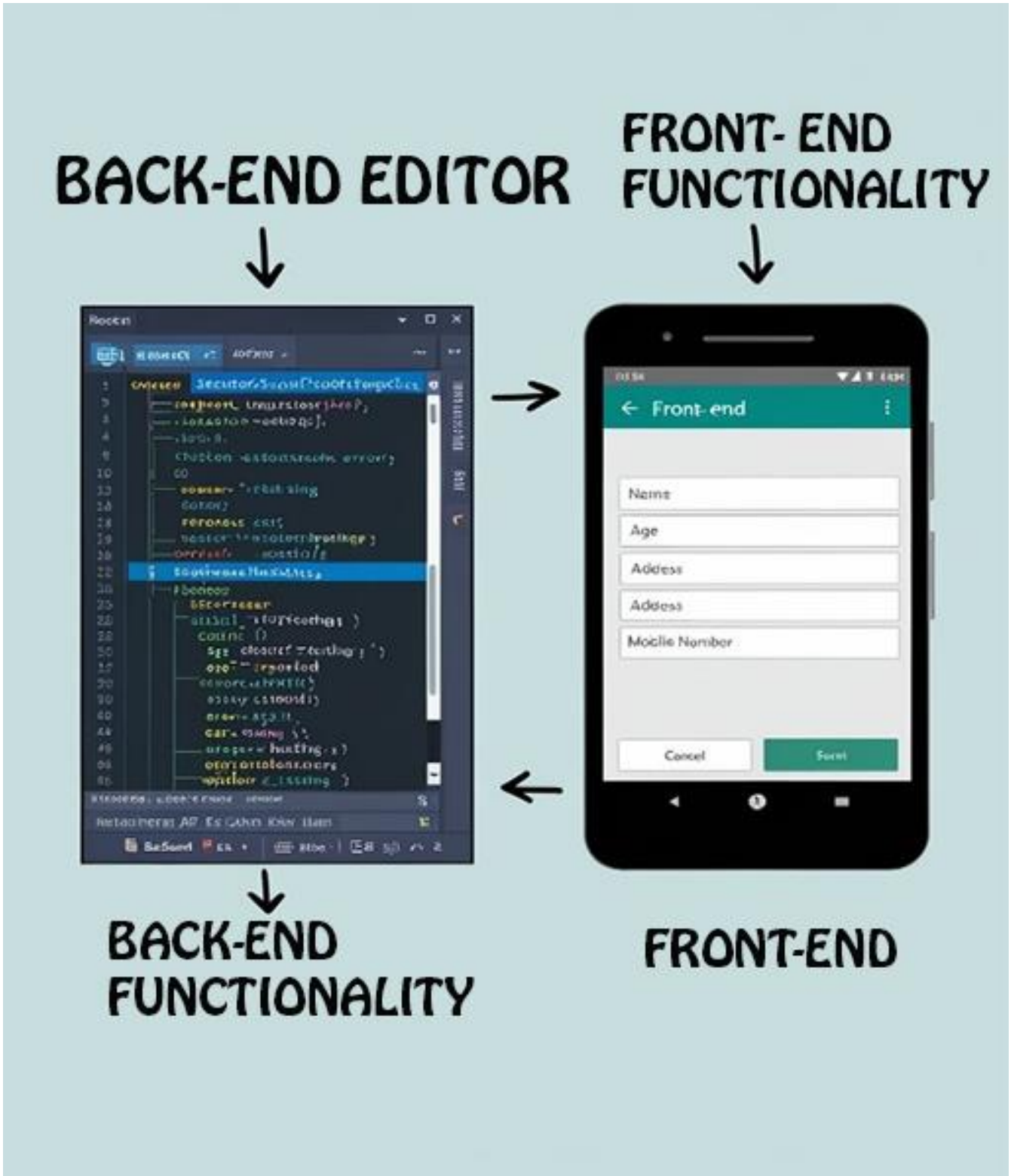
Answers	Terms	Description
1……	1. Flutter	A) The act of executing a function's code.
2……	2. Widget	B) A state management package that simplifies state management.
3……	3. GetX	C) The primary UI building block in Flutter.
4……	4. Navigator	D) The widget used to navigate between different screens.
5……	5. BLoC Pattern	E) A pattern used for managing complex state in Flutter.
		F) is a fundamental concept in object-oriented programming (OOP) that allows a class (known as a child or subclass) to inherit properties and methods from another class (known as a parent or superclass).

Practical Assessment.

TaskMaster Inc. is a startup company focused on helping users manage their tasks effectively. You, as a developer, have been assigned to create a simple task management application using Flutter. This app will allow users to create, view, complete, and delete tasks. As mobile application developer, you are requested to set up your Flutter environment, create a new project, apply core widgets, implement state management, and utilize pre-designed widgets to enhance the user interface.

References

https://www.geeksforgeeks.org/what-is-ide/
https://www.w3schools.com/
https://stackoverflow.com/
https://chatgpt.com/g/g-RGr8YRENd-chart-gpt-3
Thornton, E. (2020). Flutter For Beginners: A Genius Guide to Flutter App Development. KDP: Amazon Digital Services LLC.
Tutorials Point. (2019). Flutter. Retrieved from Tutorials Point (I) Pvt. Ltd:
Rose, R. (2022). Flutter and Dart Cookbook 1st Edition. Calfornia: O'Reilly Media.


BACK-END EDITOR
FRONT- END FUNCTIONALITY
Front-end
Name
Age
Address
Address
Mobile Number
Cancel
BACK-END FUNCTIONALITY
FRONT-END

Indicative Contents

3.1 Integration of External Services

3.2 Implement Storage Management

3.3 Implementation of Microcaps

3.4 Perform Error Handling

3.5 Perform Testing

3.6 Debug Codebase Issues.

Key Competencies for Learning Outcome 3: Integrate Backend Functionality.

Knowledge	Skills	Attitudes
• Description of external services • Description of storage management • Description of microapps • Description of Error Handling • Description of testing • Description of Codebase issues	• Performing the integration of external service • Implementing the storage management • Implementing micro apps • Performing Error Handling • Performing the testing • Debugging Codebase issues	• Having a teamwork spirit. • Being creative in creating the new flutter project. • Being adaptive. • Having Time management ability • Having good Collaboration and Communication • Having Ethical Coding

Duration: 30 hrs

Learning outcome 3 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe correctly external services as applied in Mobile Application development.
2. Perform clearly the integration of external services in flutter.
3. Implement correctly storage management based on flutter Mobile Application development.
4. Describe correctly modular microapps as applied in mobile Application Development
5. Implement clearly microapps based on integrated backend functionalities in Mobile Application Development.
6. Describe clearly Error Handling found in integrated backend functionality as applied in Mobile Application Development.
7. Perform correctly Error Handling found in integrated backend functionality as applied in Mobile Application Development.
8. Describe correctly the testing as used in flutter Mobile Application development.
9. Perform correctly the testing used in flutter Mobile Application development.
10. Describe clearly Codebase issues as applied in Mobile Application development**.**
11. Applying effectively the debugging methods as used in flutter Mobile Application development.

Resources

Equipment	Tools	Materials
● Computer (Windows, MacOS, Linux) ● Mobile Phone (Android, iPhone)	● VS Code ● Android Studio ● Xcode ● Flutter SDK. ● FlutterFlow ● Figma ● Backend software(Node.js, Laravel, Django,PHP)	● Flutter Icons ● Internet

Indicative content 3.1:Integration of external services

Duration: 5 hrs

Theoretical Activity 3.1.1: Description of external services

Tasks:

1: You are requested to answer the following questions related to the description ofexternal services:

i. Define the following terms:
   - **a.** GET
   - **b.** POST
   - **c.** PUT
   - **d.** DELETE
   - e. UPDATE
   - f. PATCH

i. Differentiate the following terms used in external service of integration Backend functionality:
   a. Adding Dependencies
   b. Adding Calls
   c. Handle Responses
   d. Authentication
   e. Authorization
   f. Firebase

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 2.3.1.

Key readings 3.1.1: Description of external services

- **Integration of External Services**

✓ **Description of HTTP Requests**

- **GET:** Used to retrieve data from a server. It is a safe and idempotent method, meaning it does not alter the state of the resource.
- **POST:** Sends data to the server to create a new resource. It can cause changes on the server and is not idempotent.
- **PUT:** Used to update an existing resource or create a new resource if it does not exist.

- It is idempotent, meaning multiple identical requests have the same effect as a single one.
- **DELETE:** Removes a resource from the server. It is idempotent, so multiple requests to delete the same resource will not cause additional effects.
- **UPDATE:** Often used interchangeably with PUT; however, it typically refers to changing an existing resource without creating a new one.
  Typically, synonymous with PUT for updating resources.
- **PATCH:** Used for partial updates to a resource. Unlike PUT, which requires the entire resource, PATCH only needs the changed fields. Partially updates an existing resource.

✓ **Adding Dependencies:** Involves including libraries or packages that allow for easier integration with external services. This is typically done through a package manager (e.g., npm for JavaScript, pubspec.yaml for Dart/Flutter).
Example: Adding dependencies in pubspec.yaml for Flutter.

```
dependencies:
  http: ^0.13.3
```

✓ **Adding Calls:** Implementing code that makes HTTP requests to external services using the dependencies added. This includes constructing the request and sending it to the server.
Example: Calling the functions defined above to interact with the API.

```
void main() async {
  await fetchUser();
  await addUser('John Doe');
  await updateUser(1, 'Jane Doe');
  await deleteUser(1);
}
```

✓ **Handle Responses:** Implementing logic to process server responses, including success and error handling. This may involve checking HTTP status codes and retrieving data from the response.
Example: Checking the status code and handling errors appropriately.

✓ **Parsing JSON Data:** Converting JSON formatted strings received from APIs into usable data structures (e.g., maps or lists) in the programming language being used.
Converting JSON strings into Dart objects.
Example:

```
import 'dart:convert';
User parseUser(String responseBody) {
  final parsed = jsonDecode(responseBody);
  return User.fromJson(parsed);
}class User {
final String name;
```

```
User({required this.name});
factory User.fromJson(Map<String, dynamic> json) {
  return User(name: json['name']);
}}
```

- ✓ **Perform Authentication and Authorization:** Implementing methods to secure API access, using techniques such as OAuth, API keys, or JWT (JSON Web Tokens) to authenticate users and authorize their access to resources.
  Securing API access using techniques such as tokens.
  Example: Using a bearer token for authorization.

```
final response = await http.get(
  Uri.parse('https://api.example.com/protected'),
  headers: {
    'Authorization': 'Bearer YOUR_TOKEN_HERE',  },);
```

- ✓ **Push Notifications (Firebase):**Integrating Firebase Cloud Messaging (FCM) to enable real-time notifications to users. This involves setting up FCM in the application and handling incoming notifications.
  Sending notifications to users using Firebase.
  Example: Setting up Firebase Cloud Messaging (FCM) in your app.
- ❖ Add Firebase to your project.
- ❖ Use the FCM SDK to send and receive messages.
- ✓ **Implement Security Measures**
- **Secure Data Storage:** Ensuring that sensitive information is stored securely, often through encryption or secure storage solutions. Use encryption for sensitive data.
- **Secure Network Communication:** Using HTTPS to encrypt data in transit, protecting it from eavesdropping and tampering. Always use HTTPS.
- **Input Validation and Output Encoding:** Implementing checks to ensure that input data is valid and appropriately sanitized to prevent security vulnerabilities like SQL injection and XSS (Cross-Site Scripting). Validate user input to prevent SQL injection and XSS attacks.

Practical Activity 3.1.2: Implementing external services and flutter framework

Task:

1. Read Key reading 3.1.2
2.Referring to the key reading 3.1.2, You are requested to go to the computer lab to Perform the integration of external service and implement security measures.
3. Present your work to the trainer/ your colleagues.
4. Ask for clarification if any.

Key readings 3.1.2: Implementing external services and flutter framework

1.Implement HTTP Requests:

Below is a comprehensive guide on how to perform various types of HTTP requests: GET, POST, PUT, DELETE, UPDATE, and PATCH.

Step 1: Add Dependency

Add the http package to your pubspec.yaml file:

Run the following command:

Step 2: Import the Package

In your Dart file, import the http package:

Step 3: Implement HTTP Requests

- **GET Request**

  To retrieve data from a server, use a GET request.

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

  if (response.statusCode == 200) {
    final data = json.decode(response.body);
    print(data); // Handle the fetched data
  } else {
    throw Exception('Failed to load data: ${response.statusCode}');
  }
}
```

- **POST Request**

  To send data to a server, use a POST request.

```
Future<void> createData(String name, String email) async {
  final response = await http.post(
    Uri.parse('https://api.example.com/data'),
    headers: {'Content-Type': 'application/json'},
    body: json.encode({'name': name, 'email': email}),
  );

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

- **PUT Request**

  To update existing data completely, use a PUT request.

```dart
Future<void> updateData(String id, String name, String email) async {
  final response = await http.put(
    Uri.parse('https://api.example.com/data/$id'),
    headers: {'Content-Type': 'application/json'},
    body: json.encode({'name': name, 'email': email}),
  );

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

- **DELETE Request**

To delete data from a server, use a DELETE request.

```dart
Future<void> deleteData(String id) async {
  final response = await http.delete(
    Uri.parse('https://api.example.com/data/$id'),
  );

  if (response.statusCode == 200) {
    print('Data deleted: ${response.body}');
  } else {
    throw Exception('Failed to delete data: ${response.statusCode}');
  }
}
```

- **PATCH Request**

To partially update existing data, use a PATCH request.

```dart
Future<void> patchData(String id, String name) async {
  final response = await http.patch(
    Uri.parse('https://api.example.com/data/$id'),
    headers: {'Content-Type': 'application/json'},
    body: json.encode({'name': name}),
  );

  if (response.statusCode == 200) {
    print('Data patched: ${response.body}');
  } else {
    throw Exception('Failed to patch data: ${response.statusCode}');
  }
}
```

2.Add dependencies

Steps to Add Dependencies in Flutter:

- ✓ **Open Your Flutter Project:** Navigate to your Flutter project directory using your terminal or your IDE.
- ✓ **Open pubspec.yaml**

  Locate the pubspec. yaml file in the root directory of your Flutter project. This file is used to manage the project's dependencies, assets, and other configurations.

- ✓ **Define Dependencies**

  Under the dependencies section, you can specify the packages you want to add. The syntax is as follows:

```
dependencies:
  flutter:
    sdk: flutter
  http: ^0.14.0  # Add the package here
```

- ✓ **Save the File**

  After adding the desired dependencies, save the pubspec. yaml file.

- ✓ **Run Flutter Packages Get**

  To install the new dependencies, run the following command in your terminal:

```
flutter pub get
```

- ✓ **Import the Package**

  After the package is installed, you can import it into your Dart files:

```
import 'package:http/http.dart' as http;
```

- ✓ **Use the Package:** Now you can use the functionalities provided by the added package in your Flutter application.

  Here's a simple example of how to add and use an HTTP package in your Flutter app:

- In pubspec.yaml:

```
dependencies:
   flutter:
      sdk: flutter
   http: ^0.14.0
```

- In a Dart file:

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

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

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

3.Adding Calls

To add functionality for making phone calls in a Flutter application, you can use the url_launcher package.

Steps to implement this:

✓ **Add Dependency**

- **Open pubspec.yaml:** Locate the pubspec.yaml file in your Flutter project.
- **Add url_launcher Package:** Under the dependencies section, add the following line:

```yaml
dependencies:
  flutter:
    sdk: flutter
  url_launcher: ^6.0.20  # Check for the latest version on pub.dev
```

- **Save the File:** Save the pubspec.yaml file.
- **Run flutter pub get:** Execute the following command in your terminal to install the new dependency:

```
flutter pub get
```

- **Import the Package**:In your Dart file where you want to initiate the call, import the package:

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

- **Create a Function to Make the Call:**

You can create a function that uses url_launcher to initiate a phone call. Here's an example:

```
void _makePhoneCall(String phoneNumber) async {
  final Uri launchUri = Uri(
    scheme: 'tel',
    path: phoneNumber,
  );
  if (await canLaunch(launchUri.toString())) {
    await launch(launchUri.toString());
  } else {
    throw 'Could not launch $launchUri';
  }
}
```

- **Use the Function**

  You can use this function in a button's onPressed callback:

```
ElevatedButton(
  onPressed: () {
    _makePhoneCall('1234567890'); // Replace with the desired phone number
  },
  child: Text('Call'),
)
```

Complete Example:

Here's a complete example of a simple Flutter app that makes a phone call:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(title: Text('Phone Call Example')),
        body: Center(
          child: ElevatedButton(
            onPressed: () {
              _makePhoneCall('1234567890'); // Replace with your phone number
            },
            child: Text('Call'),
          ),
        ),
      ),
    );
  }

  void _makePhoneCall(String phoneNumber) async {
    final Uri launchUri = Uri(
      scheme: 'tel',
      path: phoneNumber,
    );
    if (await canLaunch(launchUri.toString())) {
      await launch(launchUri.toString());
    } else {
      throw 'Could not launch $launchUri';
    }
  }
}
```

4.Handle Responses:

Here's a guide on how to handle responses in Flutter, specifically using the http package for making HTTP requests:

Step 1: Add the http Package

- **Open pubspec.yaml:** Locate the pubspec.yaml file in your Flutter project.
- **Add http Package:** Under the dependencies section, add:

```yaml
dependencies:
  flutter:
    sdk: flutter
  http: ^0.14.0  # Check for the latest version on pub.dev
```

- **Run flutter pub get**: Execute the following command to install the package:

```
flutter pub get
```

Step 2: Import the Package

- In your Dart file, import the http package:

```
import 'package:http/http.dart' as http;
import 'dart:convert'; // For JSON encoding/decoding
```

Step 3: Make an HTTP Request and Handle Responses

You can create a function to fetch data from an API and handle the response. Here's an example:

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

  if (response.statusCode == 200) {
    // If the server returns a 200 OK response, parse the JSON.
    final data = json.decode(response.body);
    print(data); // Handle the data as needed
  } else {
    // If the server returns an error response, throw an exception.
    throw Exception('Failed to load data: ${response.statusCode}');
  }
}
```

Step 4: Use the Function in Your Widget

You can call this function, for example, in the init State of a Stateful Widget or in response to a button press:

```dart
class MyHomePage extends StatefulWidget {
  @override
  _MyHomePageState createState() => _MyHomePageState();
}

class _MyHomePageState extends State<MyHomePage> {
  String _data = '';

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

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

    if (response.statusCode == 200) {
      final data = json.decode(response.body);
      setState(() {
        _data = data.toString(); // Update state with the received data
      });
    } else {
      throw Exception('Failed to load data: ${response.statusCode}');
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Handle Responses Example')),
      body: Center(
        child: Text(_data.isNotEmpty ? _data : 'Loading...'),
      ),
    );
  }
}
```

Step 6: Error Handling

You may want to handle errors gracefully. You can use a try-catch block:

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

    if (response.statusCode == 200) {
      final data = json.decode(response.body);
      setState(() {
        _data = data.toString();
      });
    } else {
      throw Exception('Failed to load data: ${response.statusCode}');
    }
  } catch (e) {
    print('Error: $e');
    setState(() {
      _data = 'Error occurred: $e';
    });
  }
}
```

5.Parsing JSON Data

Here's a step-by-step guide on how to parse JSON data effectively:

Step 1: Add the http Package

- **Open pubspec.yaml**: Locate the pubspec.yaml file in your Flutter project.
- **Add the http Package**: Under the dependencies section, add:

```yaml
dependencies:
  flutter:
    sdk: flutter
  http: ^0.14.0  # Check for the latest version on pub.dev
```

- **Run flutter pub get**: Execute the following command to install the package:

```
flutter pub get
```

Step 2: Create a Model Class

Create a model class to represent the data structure of the JSON. For example, if your JSON looks like this:

```json
{
  "id": 1,
  "name": "John Doe",
  "email": "john.doe@example.com"
}
```

You can create a Dart class:

```dart
class User {
  final int id;
  final String name;
  final String email;

  User({required this.id, required this.name, required this.email});

  // Factory constructor to create a User from JSON
  factory User.fromJson(Map<String, dynamic> json) {
    return User(
      id: json['id'],
      name: json['name'],
      email: json['email'],
    );
  }
}
```

Step 3: Fetch and Parse JSON Data

Now, you can fetch the JSON data from an API and parse it into your model class. Here's how you can do it:

```dart
import 'dart:convert';
import 'package:http/http.dart' as http;

// Function to fetch and parse user data
Future<User> fetchUser() async {
  final response = await http.get(Uri.parse('https://api.example.com/user/1'));

  if (response.statusCode == 200) {
    // If the server returns a 200 OK response, parse the JSON
    return User.fromJson(json.decode(response.body));
  } else {
    // If the server returns an error response, throw an exception
    throw Exception('Failed to load user: ${response.statusCode}');
  }
}
```

Step 4: Use the Model in Your Widget:

You can call this function in your widget, for example in the init State of a Stateful Widget:

```dart
class UserProfile extends StatefulWidget {
  @override
  _UserProfileState createState() => _UserProfileState();
}

class _UserProfileState extends State<UserProfile> {
  late Future<User> futureUser;

  @override
  void initState() {
    super.initState();
    futureUser = fetchUser();  // Fetch user data
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('User Profile')),
      body: Center(
        child: FutureBuilder<User>(
          future: futureUser,
          builder: (context, snapshot) {
            if (snapshot.connectionState == ConnectionState.waiting) {
              return CircularProgressIndicator();
            } else if (snapshot.hasError) {
              return Text('Error: ${snapshot.error}');
            } else {
              final user = snapshot.data!;
              return Column(
                mainAxisAlignment: MainAxisAlignment.center,
                children: [
                  Text('ID: ${user.id}'),
                  Text('Name: ${user.name}'),
                  Text('Email: ${user.email}'),
                ],
              );
            }
```

6.Perform Authentication and Authorization

Here's a comprehensive guide on how to implement these features using a common approach with the http package and local storage.

Step 1: Add Dependencies

First, you'll need to add some dependencies to your pubspec.yaml file:

```yaml
dependencies:
  flutter:
    sdk: flutter
  http: ^0.14.0  # For making HTTP requests
  shared_preferences: ^2.0.0  # For local storage
```

Run the following command to install the packages:

```
flutter pub get
```

Step 2: Set Up Authentication API

Assuming you have a backend API for authentication (like a REST API), the typical endpoints are:

- **POST** /login for user login.
- **POST** /register for user registration.

Step 3: Create a User Model

Create a model class for handling user data:

```
class User {
  final String token;
  User({required this.token});
}
```

Step 4: Authentication Service

Create a service for handling authentication logic:

```
import 'dart:convert';
import 'package:http/http.dart' as http;
import 'package:shared_preferences/shared_preferences.dart';

class AuthService {
  final String apiUrl = 'https://api.example.com'; // Replace with your API URL

  Future<User?> login(String username, String password) async {
    final response = await http.post(
      Uri.parse('$apiUrl/login'),
      headers: {'Content-Type': 'application/json'},
      body: json.encode({'username': username, 'password': password}),
    );

    if (response.statusCode == 200) {
      final data = json.decode(response.body);
      await _storeToken(data['token']);
      return User(token: data['token']);
    } else {
      throw Exception('Failed to login: ${response.statusCode}');
    }
  }

  Future<void> _storeToken(String token) async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.setString('auth_token', token);
  }

  Future<void> logout() async {
    final prefs = await SharedPreferences.getInstance();
    await prefs.remove('auth_token');
  }

  Future<String?> getToken() async {
    final prefs = await SharedPreferences.getInstance();
    return prefs.getString('auth_token');
  }
}
```

Step 5: Create the Login Screen

Now, create a login screen where users can enter their credentials:

```dart
import 'package:flutter/material.dart';
import 'auth_service.dart'; // Import the AuthService

class LoginScreen extends StatefulWidget {
  @override
  _LoginScreenState createState() => _LoginScreenState();
}

class _LoginScreenState extends State<LoginScreen> {
  final AuthService _authService = AuthService();
  final TextEditingController _usernameController = TextEditingController();
  final TextEditingController _passwordController = TextEditingController();
  String _errorMessage = '';

  void _login() async {
    try {
      await _authService.login(
        _usernameController.text,
        _passwordController.text,
      );
      // Navigate to home or dashboard screen
      Navigator.pushReplacementNamed(context, '/home');
    } catch (e) {
      setState(() {
        _errorMessage = e.toString();
      });
    }
  }
```

```dart
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Login')),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Column(
          children: [
            TextField(
              controller: _usernameController,
              decoration: InputDecoration(labelText: 'Username'),
            ),
            TextField(
              controller: _passwordController,
              decoration: InputDecoration(labelText: 'Password'),
              obscureText: true,
            ),
            SizedBox(height: 20),
            ElevatedButton(
              onPressed: _login,
              child: Text('Login'),
            ),
            if (_errorMessage.isNotEmpty) ...[
              SizedBox(height: 20),
              Text(_errorMessage, style: TextStyle(color: Colors.red)),
            ]
          ],
        ),
      ),
    );
  }
}
```

Step 6: Implement the Home Screen

Create a home screen that users see after logging in:

```dart
import 'package:flutter/material.dart';
import 'auth_service.dart'; // Import the AuthService

class HomeScreen extends StatelessWidget {
  final AuthService _authService = AuthService();

  void _logout(BuildContext context) async {
    await _authService.logout();
    Navigator.pushReplacementNamed(context, '/login');
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Home'),
        actions: [
          IconButton(
            icon: Icon(Icons.logout),
            onPressed: () => _logout(context),
          ),
        ],
      ),
      body: Center(child: Text('Welcome!')),
    );
  }
}
```

Step 7: Set Up Routes

Finally, set up the routes in your main.dart:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Authentication',
      initialRoute: '/login',
      routes: {
        '/login': (context) => LoginScreen(),
        '/home': (context) => HomeScreen(),
      },
    );
  }
}
```

7.Implement Push notifications(firebase)

Here's a comprehensive guide to get you started:

Step 1: Set Up Firebase

- **Create a Firebase Project**:
  - Go to the Firebase Console.
  - Click on "Add project" and follow the prompts to create a new project.
- **Add Your Flutter App**:
  - In your Firebase project, click on "Add app" and select the platform (iOS or Android).
  - Follow the instructions to register your app.
- **Download the Configuration File**:
  - For **Android**: Download google-services.json and place it in the android/app directory.
  - For **iOS**: Download GoogleService-Info.plist and add it to your Xcode project by dragging it into the Runner directory.

Step 2: Add Dependencies

Add the necessary dependencies in your pubspec.yaml:

```
dependencies:
  flutter:
    sdk: flutter
  firebase_core: ^2.0.0  # Check for the latest version
  firebase_messaging: ^14.0.0  # Check for the latest version
```

Run the following command:

```
flutter pub get
```

Step 3: Configure Android

- **Update android/build.gradle**:

Make sure to include the Google services classpath in your build.gradle file:

```
buildscript {
    dependencies {
        // Add this line
        classpath 'com.google.gms:google-services:4.3.10'  // Check for the latest versi
on
    }
}
```

- **Update android/app/build.gradle**:

At the bottom of the file, apply the Google services plugin:

```
apply plugin: 'com.google.gms.google-services'
```

- **Add Permissions**:

Ensure that your AndroidManifest.xml includes the necessary permissions and the Firebase messaging service:

```
<manifest xmlns:android="http://schemas.android.com/apk/res/android">
    <application>
        ...
        <service
            android:name="com.google.firebase.messaging.FirebaseMessagingService"
            android:exported="false">
            <intent-filter>
                <action android:name="com.google.firebase.MESSAGING_EVENT"/>
            </intent-filter>
        </service>
    </application>
</manifest>
```

Step 4: Configure iOS

- **Update ios/Podfile**:

Uncomment the platform line if necessary:

```
platform :ios, '10.0'  # Ensure this is set
```

- **Add Permissions**:

In Info.plist, add the following entries to request permission for notifications:

```
<key>UIBackgroundModes</key>
<array>
    <string>fetch</string>
    <string>remote-notification</string>
</array>
<key>FirebaseAppDelegateProxyEnabled</key>
<false/>
<key>NSUserTrackingUsageDescription</key>
<string>This app uses notifications for important updates.</string>
```

Step 5: Initialize Firebase in Flutter

In your main Dart file (usually main.dart), initialize Firebase and configure Firebase Messaging:

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

Future<void> _firebaseMessagingBackgroundHandler(RemoteMessage message) async {
  // Handle background messages
  print('Handling a background message: ${message.messageId}');
}

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter Push Notifications',
      home: NotificationHome(),
    );
  }
}
```

Step 6: Request Permissions and Handle Messages

Add the logic to request permissions and handle incoming messages:

```dart
class NotificationHome extends StatefulWidget {
  @override
  _NotificationHomeState createState() => _NotificationHomeState();
}

class _NotificationHomeState extends State<NotificationHome> {
  String _message = '';

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

  void _initFirebaseMessaging() {
    FirebaseMessaging messaging = FirebaseMessaging.instance;

    // Request permissions
    messaging.requestPermission();

    // Get the token
    messaging.getToken().then((token) {
      print("FCM Token: $token");
    });

    // Foreground messages
    FirebaseMessaging.onMessage.listen((RemoteMessage message) {
      print('Received message: ${message.notification?.title}');
      setState(() {
        _message = message.notification?.body ?? '';
      });
    });

    // Background messages
    FirebaseMessaging.onMessageOpenedApp.listen((RemoteMessage message) {
      print('Message clicked!');
    });
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Push Notifications'),
      ),
      body: Center(
        child: Text(_message.isNotEmpty ? _message : 'No new messages'),
      ),
    );
  }
}
```

Step 7: Testing Push Notifications

To test push notifications:

Use Firebase Console:

- Go to the Firebase Console.
- Navigate to Cloud Messaging.
- Click "Send a message" and fill in the fields, selecting your app as the target.

Use Postman or Curl:

You can also send messages using Postman or Curl by making a POST request to the FCM endpoint with your server key and device token.

```
curl -X POST -H "Authorization: key=YOUR_SERVER_KEY" \
-H "Content-Type: application/json" \
-d '{
  "to": "YOUR_DEVICE_TOKEN",
  "notification": {
    "title": "Test Notification",
    "body": "This is a test notification."
  }
}' "https://fcm.googleapis.com/fcm/send"
```

8.Implementing security measures

Below are steps to address secure data storage, secure network communication, and input validation/output encoding.

✓ **Secure Data Storage**

Step 1: Add Dependency

Add the flutter_secure_storage package to your pubspec.yaml:

```
dependencies:
  flutter:
    sdk: flutter
  flutter_secure_storage: ^6.0.0  # Check for the latest version
```

Run this command:

```
flutter pub get
```

Step 2: Use Secure Storage

Here's how to use flutter_secure_storage:

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

class SecureStorage {
  final _storage = FlutterSecureStorage();

  // Save data
  Future<void> saveData(String key, String value) async {
    await _storage.write(key: key, value: value);
  }

  // Read data
  Future<String?> readData(String key) async {
    return await _storage.read(key: key);
  }

  // Delete data
  Future<void> deleteData(String key) async {
    await _storage.delete(key: key);
  }
}
```

✓ **Secure Network Communication**

For secure network communication, always use HTTPS and validate SSL certificates.

Step 1: Use HTTPS

Ensure your API endpoints use HTTPS instead of HTTP. This encrypts data in transit.

```dart
final response = await http.get(Uri.parse('https://api.example.com/data'));
```

Step 2: Validate SSL Certificates

You can use the http package with a custom HttpClient to validate SSL certificates. Here's a simple example:

```dart
import 'dart:io';
import 'package:http/http.dart' as http;

class CustomHttpClient {
  static final HttpClient _httpClient = HttpClient()
    ..badCertificateCallback = (X509Certificate cert, String host, int port) => false;

  static Future<http.Response> get(String url) async {
    final request = await _httpClient.getUrl(Uri.parse(url));
    return await http.Response.fromStream(await request.close());
  }
}
```

Note: Avoid disabling certificate validation in production environments unless necessary.

✓ **Input Validation and Output Encoding**

Step 1: Input Validation

Always validate user input to prevent SQL injection, XSS, and other attacks. You can use regular expressions or built-in validation methods.

Example of validating an email:

```
bool isValidEmail(String email) {
  final regex = RegExp(r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$');
  return regex.hasMatch(email);
}
```

Step 2: Output Encoding

When displaying user input, encode it to prevent XSS attacks. You can escape HTML characters.

Here's a simple function to escape HTML entities:

```
String escapeHtml(String input) {
  return input
      .replaceAll('&', '&amp;')
      .replaceAll('<', '&lt;')
      .replaceAll('>', '&gt;')
      .replaceAll('"', '&quot;')
      .replaceAll("'", '&#39;');
}
```

The following diagrams show the working of the outputs :

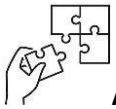
Points to Remember

GET: Fetch data from a server.

- **POST:** Send data to a server to create a resource.
- **PUT:** Update an existing resource.

- **DELETE:** Remove a resource from the server.
- **UPDATE:** (Often synonymous with PUT; may depend on context).
- **PATCH:** Partially update an existing resource.
- **Adding Dependencies**: Theoretical knowledge about dependency management and the importance of libraries or frameworks for making HTTP requests.
- **Handle Responses:** Understanding HTTP response codes and what they signify (e.g., 200 OK, 404 Not Found, 500 Internal Server Error).
- **Perform Authentication and Authorization:** Knowledge of various authentication methods (e.g., OAuth, API keys) and their importance in securing API interactions.
- **Understanding security practices like:** Secure Data Storage, Secure Network Communication, Input Validation and Output Encoding.
- **Adding calls:** Understanding API calls and their structure.
- **Security measures:** The ways of implementing the security measures are Secure Data Storage, Secure Network Communication and Input Validation and Output Encoding.

  While performing the integration of external services pass through the following steps:
- **Implementing HTTP Requests:**in code, using tools like Postman or cURL, and handling responses.
- **Adding Dependencies:** Using package managers (like npm, Maven, or Gradle) to add libraries to a project.
- **Adding Calls:** Writing code to make API calls, handle errors, and manage request lifecycle.
- **Handle Responses:** Writing code to handle and process responses, including error handling.
- **Parsing JSON Data:** Implementing code to parse JSON in different programming languages 6.
- **Perform Authentication and Authorization:** Implementing authentication flows in an application.
- **Push Notifications (Firebase):** Setting up Firebase Cloud Messaging in an app and sending notifications.
- **While Implementing Security Measures pass through the following steps:**
- **Secure Data Storage:** Implementing secure storage techniques (e.g., using Keychain in iOS, Encrypted Shared Preferences in Android).
- **Secure Network Communication:** Implementing HTTPS and configuring secure connections in applications.
- **Input Validation and Output Encoding:** Writing validation functions and encoding output to prevent security issues.



Application of learning 3.1.

You are developing a mobile application for a fitness company called **Maranata Primary school** located in **RUBAVU** district. The app allows users to create an account, log their workouts, and receive motivational push notifications. Your task is to implement the registration feature that includes the following requirements: to implement HTTP Requests using tools like Postman or cURL, and handling responses, adding dependencies, Adding Calls to Integrate an API call to register a new user with **Maranata Primary's** backend, Handling Responses to Ensure the application properly handles the response from the registration API, Parsing JSON Data to Extract necessary information from the JSON response such as the user ID and authentication token, Authentication and Authorization to Implement secure authentication methods to protect user data, Push Notifications (Firebase) to Set up Firebase push notifications for sending a welcome message to users upon successful registration and Incorporate security measures to protect user data both during transmission and while stored

Indicative content 3.2 Implement storage management

Duration: 4 hrs

Theoretical Activity 3.2.1: Description of storage management

Tasks:

1: You are requested to answer the following questions related to the description of storage management:

i. Explain the following terms:

a. Data Integrity
b. Security Standards
c. Differentiate the following Use Local Data Storage of an integrate Backend functionality:
d. Working with Shared Preferences
e. Working with SQLite
f. Working with File Storage

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class.

4: For more clarification, read the key readings 3.2.1. In addition, ask questions where necessary.

Key readings 3.2.1: Description of storage management

- ✓ **Data Integrity:**Ensuring that the data remains accurate, consistent, and reliable throughout its lifecycle. This involves implementing validation and error-checking mechanisms.Ensuring that data remains accurate and consistent throughout its lifecycle.
  Example: Implementing checks to validate data before saving it.
- ✓ **Security Standards:**Adhering to established security practices and regulations to protect data from unauthorized access and breaches. This includes compliance with standards such as GDPR or HIPAA. Adhering to best practices to protect data from unauthorized access.
  Example: Using encryption for stored data.
- ✓ **Use Local Data Storage**
- **Working with Shared Preferences:** Using shared preferences for storing small amounts of key-value data, often used for user preferences and settings.
  Example:

```
import 'package:shared_preferences/shared_preferences.dart';
Future<void> saveUserPreference(String key, String value) async {
 final prefs = await SharedPreferences.getInstance();
 await prefs.setString(key, value);
}
Future<String?> getUserPreference(String key) async {
 final prefs = await SharedPreferences.getInstance();
 return prefs.getString(key);
```

- **Working with SQLite:**Implementing SQLite for structured data storage, allowing for complex queries and transactions. This is suitable for applications requiring relational database capabilities.

```
Example:
import 'package:sqflite/sqflite.dart';
import 'package:path/path.dart';
Future<Database> initializeDB() async {
 String path = join(await getDatabasesPath(), 'users.db');
 return openDatabase(path, onCreate: (database, version) async {
  await database.execute(
   "CREATE TABLE users(id INTEGER PRIMARY KEY AUTOINCREMENT, name TEXT)",
  );
 }, version: 1);
}
Future<void> insertUser(User user) async {
 final db = await initializeDB();
 await db.insert('users', user.toMap());
}
```

- **Working with File Storage:**Handling data storage in files on the device, which can include reading and writing text or binary files for various purposes.

```
Example:
import 'dart:io';
import 'package:path_provider/path_provider.dart';
Future<File> get localFile async {
 final path = await getApplicationDocumentsDirectory();
 return File('${path.path}/data.txt');
}
Future<void> writeData(String data) async {
 final file = await localFile;
 await file.writeAsString(data);}
```

Practical Activity 3.2.2: Implementing storage management

Task:

1: Read Key reading 3.2.2

2:Referring to the key reading 3.2.2, You are requested to go to the computer lab to Implement storage management

3:Present your work to the trainer/ your colleagues.

4:Ask for clarification if any.

Key readings3.2.2: Implementing storage management

Comprehensive guide to achieve this:

1. Implement Data Integrity

Data integrity ensures that the data is accurate and consistent. Here are some strategies to implement data integrity:

- **Validation**: Validate data before saving it. For example, ensure that required fields are not empty and that data formats are correct.

```
bool isValidEmail(String email) {
  final regex = RegExp(r'^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$');
  return regex.hasMatch(email);
}
```

- **Error Handling**: Handle errors when performing database operations, such as inserting or updating records.

```
try {
  // database operation
} catch (e) {
  // handle error
}
```

- **Transactions**: Use database transactions where applicable to ensure that a group of operations either all succeed or all fail.

2. Apply Security Standards

To secure stored data, you can take the following measures:

- **Encryption**: Encrypt sensitive data before saving it. You can use packages like encrypt for this purpose.

```dart
import 'package:encrypt/encrypt.dart' as encrypt;

final key = encrypt.Key.fromUtf8('32charactertestkey12345678901234'); // 32 character key
final iv = encrypt.IV.fromLength(16); // Initialization vector

String encryptData(String data) {
  final encrypter = encrypt.Encrypter(encrypt.AES(key));
  return encrypter.encrypt(data, iv: iv).base64;
}

String decryptData(String encryptedData) {
  final encrypter = encrypt.Encrypter(encrypt.AES(key));
  return encrypter.decrypt64(encryptedData, iv: iv);
}
```

- **Secure Storage**: Use flutter_secure_storage for sensitive data like tokens and passwords.

3. Working with Shared Preferences

Shared Preferences is useful for storing simple data in key-value pairs.

Step 1: Add Dependency

Add shared_preferences to your pubspec.yaml:

```yaml
dependencies:
  shared_preferences: ^2.0.0  # Check for the latest version
```

Run:

```
flutter pub get
```

Step 2: Using Shared Preferences

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

// Save data
Future<void> saveData(String key, String value) async {
  final prefs = await SharedPreferences.getInstance();
  await prefs.setString(key, value);
}

// Read data
Future<String?> readData(String key) async {
  final prefs = await SharedPreferences.getInstance();
  return prefs.getString(key);
}

// Remove data
Future<void> removeData(String key) async {
  final prefs = await SharedPreferences.getInstance();
  await prefs.remove(key);
}
```

4. Working with SQLite

SQLite is suitable for structured data storage.

Step 1: Add Dependency

Add sqflite and path to your pubspec.yaml:

```yaml
dependencies:
  sqflite: ^2.0.0+4  # Check for the latest version
  path: ^1.8.0  # For file path handling
```

Run:

```
flutter pub get
```

Step 2: Create a Database Helper

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

class DatabaseHelper {
  static final DatabaseHelper instance = DatabaseHelper._init();

  static Database? _database;

  DatabaseHelper._init();

  Future<Database> get database async {
    if (_database != null) return _database!;
    _database = await _initDB('app.db');
    return _database!;
  }

  Future<Database> _initDB(String filePath) async {
    final dbPath = join(await getDatabasesPath(), filePath);
    return await openDatabase(dbPath, version: 1, onCreate: _createDB);
  }

  Future _createDB(Database db, int version) async {
    await db.execute('''
      CREATE TABLE users(
        id INTEGER PRIMARY KEY AUTOINCREMENT,
        name TEXT,
        email TEXT
      )
    ''');
  }
```

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

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

5. **Working with File Storage**

For more complex data or files (like images), you might want to use file storage.

Step 1: Add Dependency

Add path_provider to your pubspec.yaml:

```
dependencies:
  path_provider: ^2.0.0  # Check for the latest version
```

Run:

```
flutter pub get
```

Step 2: Using File Storage

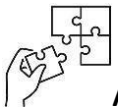
```dart
import 'dart:io';
import 'package:path_provider/path_provider.dart';

// Get the directory for file storage
Future<String> get _localPath async {
  final directory = await getApplicationDocumentsDirectory();
  return directory.path;
}

// Write to a file
Future<File> writeData(String fileName, String data) async {
  final path = await _localPath;
  return File('$path/$fileName.txt').writeAsString(data);
}

// Read from a file
Future<String> readData(String fileName) async {
  try {
    final path = await _localPath;
    return await File('$path/$fileName.txt').readAsString();
  } catch (e) {
    return 'Error: $e';
  }
}
```

Points to Remember

Data Integrity: Understanding the principles and importance of data integrity, including concepts like consistency, accuracy, and reliability of stored data.
Security Standards: Knowledge of security standards and best practices for data storage, including encryption methods, access controls, and compliance with regulations (e.g., GDPR, HIPAA).

Local Data Storage: Local data storage is Shared Preferences, SQLite, File Storage
While implementing the Storage Management pass through the following steps:

- Implement data integrity.
- Apply Security standards.
- Implement the Working with Shared Preferences.
- Create the Working with SQLite**.**
- Implement the Working with File Storage.



Application of learning 3.2

You are developing a mobile application for a health and wellness company called **WellnessTrack**. The app allows users to track their daily water intake, save their favourite healthy recipes, and store user preferences for a personalized experience. Your task is to Implement data integrity, Apply Security standards, implement local data storage features using Shared Preferences, SQLite, and File Storage. you are requesting to implement local data storage in the WellnessTrack app to achieve the requirements of Storing the user's daily water intake goal (a simple integer value) using Shared Preferences, SQLite, Allowing users to upload and store images of their meals in the app using File Storage.

Indicative content 3.3: Implementation of Microapps

Duration: 12 hrs

Theoretical Activity 3.3.1: Description of microapps

Tasks:

1: You are requested to answer the following questions related to the description ofmicroapps:

i. Define the following terms:
   a. Modular Microapp
   b. Project Structure
   c. Dependency Injection
   d. Shared Components

ii. Differentiate Android gradle fromiOS build settingsused in Microapp Build Configuration

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class.

4: For more clarification, read the key readings 3.3.1. In addition, ask questions where necessary.

Key readings 3.3.1: Implementation of microapps

✓ **Description of Modular Microapp**

- **Definition:** Microapps are small, independently deployable applications that focus on specific functionalities. They allow for a modular approach to app development, enhancing flexibility and scalability.
- **Structure:** Each microapp typically consists of its own codebase, resources, and dependencies. They communicate with each other through APIs or messaging systems.

✓ **Applying Modular Microapps Concept**

- **Project Structure:**Organizing the project in a way that reflects modularity, ensuring that each microapp is self-contained and can be developed, tested, and deployed independently.
- **Dependency Injection:**Implementing dependency injection to manage dependencies between microapps, facilitating loose coupling and easier testing.
- **Shared Components:**Identifying and building shared components (e.g., libraries, utilities) that can be reused across different microapps to reduce redundancy.

✓ **Perform Microapp Build Configuration**

- **Configure Each Module Pubspec File:**
  Setting up the pubspec.yaml file for Dart/Flutter modules to manage dependencies, versioning, and metadata.

```
# Example pubspec.yaml for a microapp
name: my_microapp
dependencies:
  flutter:
    sdk: flutter
  http: ^0.13.3
```

- **Android Gradle:**Configuring Gradle build files for Android modules, specifying dependencies, build types, and versioning.

```
// Example build.gradle for a microapp module
dependencies {
    implementation 'com.android.support: appcompat-v7:28.0.0'
}
```

- **iOS Build Settings:** Adjusting Xcode build settings for iOS modules, ensuring proper.
  integration and configuration for building the application on iOS devices.
  Configure settings in Xcode for each microapp module, ensuring proper integration.

Practical Activity 3.3.2: Implementing microapps

Task:

1. Read Key reading 3.3.2
2.Referring to the key reading 3.2.2, You are requested to go to the computer lab to Implement microapps.
3. Present your work to the trainer/ your colleagues.
4. Ask for clarification if any.

Key readings 1.1.2Key readings 3.3.2:Implement microapps

1.Implement modular microapp

Here's a step-by-step guide to implementing a microapp using the modular microapps concept.

Step 1: Set Up Your Flutter Environment

Ensure you have Flutter installed and set up on your machine. You can verify the installation by running:

```
flutter doctor
```

Step 2: Create a New Flutter Project

Create a new Flutter project that will serve as your microapp.

```
flutter create microapp_demo
cd microapp_demo
```

Step 3: Organize Your Project Structure

Organize your project into modules. For example, you might create two modules: authentication and dashboard.

```
microapp_demo/
├── lib/
│   ├── main.dart
│   ├── modules/
│   │   ├── authentication/
│   │   │   ├── login.dart
│   │   │   ├── register.dart
│   │   └── dashboard/
│   │       ├── dashboard.dart
│   │       └── widgets/
│   │           └── dashboard_widget.dart
```

Step 4: Create the Authentication Module

- **Create Login and Register Screens**:

In lib/modules/authentication/login.dart:

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

class LoginScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Login')),
      body: Center(
        child: Text('Login Screen'),
      ),
    );
  }
}
```

In lib/modules/authentication/register.dart:

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

class RegisterScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Register')),
      body: Center(
        child: Text('Register Screen'),
      ),
    );
  }
}
```

A

dd Navigation:

In lib/main.dart, set up navigation to the authentication module:

```dart
import 'package:flutter/material.dart';
import 'modules/authentication/login.dart';
import 'modules/authentication/register.dart';

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Microapp Demo',
      initialRoute: '/',
      routes: {
        '/': (context) => HomePage(),
        '/login': (context) => LoginScreen(),
        '/register': (context) => RegisterScreen(),
      },
    );
  }
}
```

```
class HomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Home')),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            ElevatedButton(
              onPressed: () => Navigator.pushNamed(context, '/login'),
              child: Text('Login'),
            ),
            ElevatedButton(
              onPressed: () => Navigator.pushNamed(context, '/register'),
              child: Text('Register'),
            ),
          ],
        ),
      ),
    );
  }
}
```

Step 5: Create the Dashboard Module

- **C**

reate Dashboard Screen:

In lib/modules/dashboard/dashboard.dart:

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

class DashboardScreen extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Dashboard')),
      body: Center(
        child: Text('Dashboard Screen'),
      ),
    );
  }
}
```

- **A**

dd Navigation to Dashboard:

Update the home page in lib/main.dart to navigate to the dashboard:

```
// Add this button in HomePage
ElevatedButton(
  onPressed: () => Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => DashboardScreen()),
  ),
  child: Text('Dashboard'),
),
```

Step 6: Modular Microapps Concept

To apply the modular microapps concept:

- **Encapsulation**: Each module (authentication and dashboard) should handle

its own state and logic. You can use packages like provider or bloc for state management within each module.

- **Independent Development**: Each module can be developed independently. You can create separate repositories for each module if needed. As shown on the following source code:

```
final getIt = GetIt.instance;

void setup() {
  getIt.registerLazySingleton<SomeService>(() => SomeServiceImpl());
}

class SomePage extends StatelessWidget {
  final someService = getIt<SomeService>();

  @override
  Widget build(BuildContext context) {
    return Container(); // Use the service
  }
}
```

Step 7: Perform Microapp Build Configuration

- **Configure Build Settings**:

You can use the Flutter build configuration to create different environments (e.g., development, production) and enable or disable specific modules.

In your pubspec.yaml, you can define different flavors if needed. Here's a simple example using Dart defines:

```
flutter:
  ...
```

- **C**

onfigure Each Module Pubspec File:

```
# Example pubspec.yaml for a microapp
name: my_microapp
dependencies:
  flutter:
    sdk: flutter
  http: ^0.13.3
```

- A

ndroid Gradle

```
// Example build.gradle for a microapp module
dependencies {
    implementation 'com.android.support:appcompat-v7:28.0.0'
}
```

- **B**

uild for Different Environments:

Use different build commands to create builds for different configurations:

```
flutter run --release --dart-define=FLAVOR=production
```

Step 8: Testing and Debugging

- Ensure you thoroughly test each module independently.

- Use unit tests and widget tests to verify the functionality of each microapp.

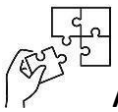
Points to Remember

- **Microapps:** are small, independently deployable applications focusing on specific functionalities.
- **Project Structure:** Organize projects to reflect modularity, allowing each microapp to be self-contained for independent development, testing, and deployment.
- **Dependency Injection:** Use dependency injection to manage inter-microapp dependencies, promoting loose coupling and easier testing.
- Shared Components: Identify and create shared components (e.g., libraries, utilities).
- **Configure Each Module Pubspec File:** Set up the pubspec.yaml file for Dart/Flutter modules to manage dependencies, versioning, and metadata.
- **Android Gradle:** Configure Gradle build files for Android modules, detailing dependencies, build types, and versioning.

  iOS Build Settings: Adjust Xcode build settings for iOS modules to ensure proper integration and configuration for building the application on iOS devices.

  While implementing the Storage Management pass through the following steps:

- Create a simple Microapp.
- Apply Modular Microapps Concept
- Perform Microapp Build Configuration.



Application of learning 3.3.

You are part of the development team for **TravelGenie,** a travel planning application that allows users to create personalized itineraries, book accommodations, and explore local attractions. To enhance the app's scalability and maintainability, the team has decided to adopt Microapp architecture. Your tasks involve of creating a simple microapp, applying the Modular Microapps concept, and performing Microapp build configuration in the **TravelGenie** application.

Indicative content 3.4: Perform Error Handling

Duration: 4 hrs

Theoretical Activity 3.4.1: Description of Error Handling

Tasks:

1: You are requested to answer the following questions related to the description of Error Handling:

i. Define the following terms:
   a. Error Handling
   b. try-catch Block
   c. OnError
   d. CatchError

ii. List the 3 types of error classes.

iii. Differentiate Finally Block from onClause of Rethrowing Exceptions error handling.

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class.

4: For more clarification, read the key readings 3.4.1. In addition, ask questions where necessary.

Key readings 3.4.1: Perform Error Handling

✓ **Description**

Definition of Error Handling:

Error handling is the process of anticipating and managing errors that may occur during the execution of a program. It enhances the robustness of applications by allowing developers to respond to errors gracefully instead of crashing the application.

Error Classes: Errors can be categorized into various classes:

- **Syntax Errors:** Mistakes in the code that violate the language's grammar.
- **Runtime Errors:** Errors that occur during the execution of the program, such as dividing by zero.
- **Logical Errors:** Flaws in the program logic that produce incorrect results.

✓ **Exception Management**

- **try-catch Block:** A fundamental structure for handling exceptions. Code that might throw an exception is placed inside a try block, and the corresponding catch block contains code that executes if an exception occurs, allowing for recovery or logging.

  Example:

```
try {
  int result = 10 ~/ 0; // This will throw an exception
} catch (e) {
  print('Caught an exception: $e');
}
```

- **OnError:** A specific method to handle errors that allows for defining custom error-handling logic. It can be used to catch errors that occur in asynchronous operations.

  Example: Used in stream handling to catch errors.

```
Stream<int>.periodic(Duration(seconds: 1), (x) => x)
    .handleError((error) {
      print('Caught error: $error');
    });
```

- **CatchError:**Similar to try-catch, but specifically used for handling errors in streams or asynchronous operations. It allows developers to respond to errors in a more controlled manner.

  Example:

  Similar to onError, but specifically for handling errors in futures.

```
fetchUser().catchError((error) {
  print('Error fetching user: $error');
});
```

✓ **Rethrowing Exceptions**

- **Finally Block:** A block that follows the try and catch blocks. Code within the finally block executes regardless of whether an exception was thrown or caught, typically used for cleanup operations.

  Example:

```
try {
  // Some code that may throw
} catch (e) {
  print('Error: $e');
} finally {
  print('This always runs, regardless of error.');
}
```

- **onClause:** Used in conjunction with try-catch to handle specific types of exceptions. This allows for more granular error handling, enabling different responses based on the exception type.

Example: To catch specific exceptions.

```
try {
  throw FormatException('Invalid format');
} on FormatException catch (e) {
  print('Caught a format exception: $e');
}
```

Practical Activity 3.4.2: Perform Error Handling

Task:

1: Read Key reading 3.4.2

2:Referring to the key reading 3.2.2, You are requested to go to the computer lab to Perform Error Handling

3: Present your work to the trainer/ your colleagues.

4: Ask for clarification if any.

Key readings 3.4.2: Perform Error Handling

Here's how to implement various error handling techniques in Flutter, including try-catch blocks, onError, catchError, finally blocks, and on clauses.

Step 1: Set Up Your Flutter Project

If you haven't already, create a new Flutter project:

```
flutter create error_handling_demo
cd error_handling_demo
```

Step 2: Implement Try-Catch Block

The try-catch block allows you to catch exceptions that occur within the try block.

Example:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Error Handling Demo',
      home: ErrorHandlingScreen(),
    );
  }
}

class ErrorHandlingScreen extends StatelessWidget {
  void riskyOperation() {
    try {
      // Simulate an error
      throw Exception('An error occurred!');
    } catch (e) {
      print('Caught an error: $e');
    }
  }
```

```dart
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Error Handling')),
      body: Center(
        child: ElevatedButton(
          onPressed: riskyOperation,
          child: Text('Trigger Error'),
        ),
      ),
    );
  }
}
```

Step 3: Use OnError

OnError is typically used in asynchronous operations, often with Futures.

Example:

```dart
Future<void> asyncOperation() async {
  return Future.error('An async error occurred!');
}

void handleAsyncOperation() {
  asyncOperation().catchError((error) {
    print('Caught async error: $error');
  });
}
```

Call handleAsyncOperation() in the onPressed of your button.

Step 4: Use CatchError

catchError can be used with Futures to handle errors directly.

Example:

```dart
Future<void> anotherAsyncOperation() async {
  return Future.error('Another async error!');
}

void handleAnotherAsyncOperation() {
  anotherAsyncOperation().catchError((error) {
    print('Caught another async error: $error');
  });
}
```

Step 5: Use Finally Block

A final block executes code after the try and catch blocks, regardless of whether an error occurred.

Example:

```
void riskyOperationWithFinally() {
  try {
    // Simulate an error
    throw Exception('Error in operation!');
  } catch (e) {
    print('Caught an error: $e');
  } finally {
    print('This will always execute!');
  }
}
```

Call riskyOperationWithFinally() in the onPressed of your button.

Step 6: Use On Clause

The on clause is useful for catching specific exception types.

Example:

```
void specificErrorHandling() {
  try {
    // Simulate a specific error
    throw FormatException('This is a format error!');
  } on FormatException catch (e) {
    print('Caught a format error: $e');
  } catch (e) {
    print('Caught a general error: $e');
  }
}
```

Call specificErrorHandling() in the onPressed of your button.

Combined Example.

Here's how all these error handling techniques can be put together in a single Flutter app:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Error Handling Demo',
      home: ErrorHandlingScreen(),
    );
  }
}

class ErrorHandlingScreen extends StatelessWidget {
  void riskyOperation() {
    try {
      throw Exception('An error occurred!');
    } catch (e) {
      print('Caught an error: $e');
    }
  }

  Future<void> asyncOperation() async {
    return Future.error('An async error occurred!');
  }
```

```
void riskyOperationWithFinally() {
  try {
    throw Exception('Error in operation!');
  } catch (e) {
    print('Caught an error: $e');
  } finally {
    print('This will always execute!');
  }
}

void specificErrorHandling() {
  try {
    throw FormatException('This is a format error!');
  } on FormatException catch (e) {
    print('Caught a format error: $e');
  } catch (e) {
    print('Caught a general error: $e');
  }
}
```

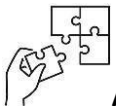
```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text('Error Handling')),
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          ElevatedButton(
            onPressed: riskyOperation,
            child: Text('Try-Catch'),
          ),
          ElevatedButton(
            onPressed: handleAsyncOperation,
            child: Text('Async Operation with CatchError'),
          ),
          ElevatedButton(
            onPressed: riskyOperationWithFinally,
            child: Text('Try-Catch with Finally'),
          ),
          ElevatedButton(
            onPressed: specificErrorHandling,
            child: Text('Specific Error Handling'),
          ),
        ],
      ),
    ),
  );
 }
}
```

Points to Remember

- **Error Handling:** Understanding the concept of error handling, its importance in software development, and how it contributes to robustness and reliability.
- **Error Classes:** Knowledge of different types of errors (e.g., syntax errors, runtime errors, logical errors) and how they can be categorized in programming languages.
- **try-catch Block:** Theoretical understanding of how try-catch blocks work, their purpose, and the flow of control during exception handling.
- **OnError:** Learning about the OnError mechanism, its usage, and contexts where it might be applicable.
- **CatchError:** Understanding the CatchError method and its role in handling specific types of errors.
- **Finally Block:** Theoretical knowledge about the finally block, its purpose in ensuring that certain code executes regardless of whether an exception occurred.
- **on Clause:** Understanding how the on clause can be used to catch specific exceptions and its role in exception filtering.

While Performing error handling pass through the following steps:

- Perform try-catch Block.
- Perform OnError.
- Perform CatchError.
- Perform Finally Block.
- Perform onClause



Application of learning 3.4.

You are a software engineer at **HealthTrack**, a mobile application designed to help users monitor their fitness and health metrics. As the app continues to grow, you need to ensure that it handles errors gracefully to maintain a seamless user experience. Your tasks involve of implementing error handling, using try-catch Block, OnError, CatchError, Finally Block, and onClause in the **HealthTrack** application.

Indicative content 3.5: Perform Testing

Duration: 3 hrs

Theoretical Activity 3.5.1: Description of Testing

Tasks:

1: You are requested to answer the following questions related to the description ofTesting:

i. Define the following terms:
   a. Unit Tests
   b. Widget Tests
   c. Integration Tests
   d. Functional Tests
   e. UI Tests
   f. Performance Tests
   g. Regression Tests
   h. Cross-Platform Testing
   i. Security Testing
   j. End-to-End (E2E) Tests
   k. Mocking and Stubbing
   l. Code Coverage Tests

ii. Differentiate the following terms used in Test Reliability of integrating Backend functionality:
   a. Rethrowing Exceptions
   b. Unit Testing
   c. Integration Testing
   d. System Testing
   e. Acceptance Testing.

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class.

4: For more clarification, read the key readings 3.5.1. In addition, ask questions where necessary.

Key readings3.5.2: Description of Testing

✓ **Description**

- **Definition:**

  Testing is the process of evaluating a system or its components to determine whether it satisfies the specified requirements and functions correctly. It aims to identify bugs and ensure the quality of the software.

- **Importance:**

  Testing is crucial for:

  Ensuring software reliability and performance.

  Reducing maintenance costs by identifying issues early.

  Enhancing user satisfaction by delivering a functional product.

- **Testing Levels:**

  Different levels of testing include:

❖ **Unit Testing:** Tests individual components or functions in isolation.

❖ **Integration Testing:** Tests the interaction between integrated components.

❖ **System Testing:** Tests the complete system as a whole.

❖ **Acceptance Testing:** Validates the end-to-end business flow.

✓ **Implement Types of Testing**

- **Unit Tests:**Focus on testing small units of code (e.g., functions or classes) to ensure they work as intended.

  Example:

```
import 'package:test/test.dart';
void main() {
  test('String split', () {
    var str = 'hello world';
    expect(str.split(' '), ['hello', 'world']);
  });
}
```

- **Widget Tests:**Specifically for testing UI components in frameworks like Flutter, verifying that widgets behave correctly.

  Example:

```
import 'package:flutter_test/flutter_test.dart';
void main() {
  testWidgets('Counter increments smoke test', (WidgetTester tester) async {
    await tester.pumpWidget(MyApp());
    expect(find.text('0'), findsOneWidget);
    await tester.tap(find.byIcon(Icons.add));
    await tester.pump();
    expect(find.text('1'), findsOneWidget);
```

```
  });
}
```

- **Integration Tests:**Ensure that different modules or services work together correctly.

Example:

```
import 'package:flutter_driver/flutter_driver.dart';
import 'package:test/test.dart';
void main() {
  group('App integration test', () {
    final buttonFinder = find.byValueKey('increment');
    FlutterDriver driver;
    setUpAll(() async {
      driver = await FlutterDriver.connect();
    });
    tearDownAll(() async {
      if (driver != null) {
        await driver.close();
      }
    });
    test('increments the counter', () async {
      await driver.tap(buttonFinder);
      // Additional checks...
    });
  });
}
```

- **Functional Tests:**Assess specific functionalities or features of the application against the requirements.
- **UI Tests:**Verify the user interface to ensure it behaves as expected from a user's perspective.
- **Performance Tests:**Evaluate the application's responsiveness, stability, and scalability under a particular workload.
- **Regression Tests:**Ensure that new code changes do not adversely affect existing functionality.
- **Cross-Platform Testing:**Tests the application across different platforms (e.g., web, mobile) to ensure consistent behavior.
- **Security Testing:**Identify vulnerabilities and ensure that the application protects data and maintains functionality as intended.
- **End-to-End (E2E) Tests:**Test the entire application flow from start to finish to ensure everything works together as expected.
- **Mocking and Stubbing:**Techniques used in testing to simulate the behavior of complex objects or systems, allowing for isolated testing of components.

- **Code Coverage Tests:**Measure the percentage of code that has been executed during testing, helping to identify untested paths.

✓ **Test Device Responsiveness**

- **Select Testing Tools:**Choosing appropriate testing frameworks and tools based on the project requirements and technology stack.
- **Test Using Emulator and Simulator:**Using emulators (for Android) and simulators (for iOS) to replicate the behavior of devices without physical hardware.
- **Test Using Physical Device**:Conducting tests on actual devices to evaluate real-world performance and usability.
- **Perform Manual Testing:**Testing the application manually to identify issues that automated tests may miss.
- **Perform Automated Testing:**Using scripts and tools to automate the execution of tests, improving efficiency and coverage.

✓ **Test Reliability**

- **Unit and Widget Testing:**Focus on ensuring that individual components and UI elements function reliably under various conditions.
- **Integration and End-to-End Testing:**Validate the reliability of interactions between components and the overall workflow of the application.
- **Edge Case and Stress Testing:**Testing how the application behaves under unusual or extreme conditions to ensure stability.
- **Performance Testing and User Acceptance Testing (UAT):**
  Evaluate the application's performance under load and gather feedback from end-users to confirm that it meets their needs.

Practical Activity 3.5.2: Perform Testing

Task:

1:Read Key reading 3.5.2

2:Referring to the key reading 3.5.2, You are requested to go to the computer lab to Perform Testing

3:Present your work to the trainer/ your colleagues.

4:Ask for clarification if any.

Key readings 3.5.2: Perform Testing

1. Unit Tests

Steps:

- **Add Dependencies**:

Add the flutter_test package (included by default in Flutter projects).

- **Create Test Files**:

Create a new directory called test (if it doesn't exist) and add your test files.

- **Write Unit Tests**:

Example of a simple unit test:

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

int add(int a, int b) => a + b;

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

- **Run Tests**:

Use the command:

```
flutter test
```

2. Widget Tests

Widget tests verify the functionality and appearance of a single widget.

Steps:

- **Write Widget Tests**:

Example of a widget test:

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

class MyWidget extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Text('Hello, World!');
  }
}

void main() {
  testWidgets('MyWidget has a text "Hello, World!"', (WidgetTester tester) async {
    await tester.pumpWidget(MaterialApp(home: MyWidget()));
    expect(find.text('Hello, World!'), findsOneWidget);
  });
}
```

- **Run Tests**:

  Use the command:

```
flutter test
```

3. Integration Tests

Integration tests verify that multiple widgets and services work together.

Steps:

- **Add Dependencies**:

  Update pubspec.yaml to include integration_test:

```yaml
dev_dependencies:
  integration_test:
    sdk: flutter
```

- **Create Integration Test File**:

  Create a directory integration_test and add your test file.

- **Write Integration Tests**:

  Example of an integration test:

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

void main() {
  IntegrationTestWidgetsFlutterBinding.ensureInitialized();

  testWidgets('Integration Test Example', (WidgetTester tester) async {
    await tester.pumpWidget(MyApp());
    expect(find.text('Hello, World!'), findsOneWidget);
  });
}
```

- **Run Tests**:

  Use the command:

```
flutter test integration_test
```

4.Testing Tools.

Select one of the following Testing Tools according to the type of test that you want to implement.

- **Flutter Test**: Built-in testing framework for unit and widget tests.
- **Mockito**: Used for mocking dependencies in tests.
- **Integration Test**: For running integration tests.
- **Flutter Driver**: For end-to-end tests (deprecated in favour of integration_test).

5. Test Using Emulator and Simulator

- **Run Emulator**:

  Start an emulator using Android Studio or from the command line:

```
flutter emulators --launch <emulator_id>
```

- **Run Tests**:

  Execute tests in the emulator:

```
flutter test
```

6. Test Using Physical Device

- **Connect Device:**

  Connect your physical device via USB and ensure USB debugging is enabled.

- **Run Tests**:

  Execute tests directly on the device:

```
flutter test
```

7. Manual Testing

- **Deploy Application**:

  Build and run your app on a device or emulator.

- **Interact with UI**:

  Manually navigate through the app and check for expected behaviors.

8. Automated Testing

- **Setup CI/CD**:

  Use tools like GitHub Actions, Travis CI, or CircleCI to automate running tests on every commit.

- **Write Tests**:

  Ensure comprehensive test coverage, including unit, widget, and integration tests.

9. E2E Tests

End-to-end tests simulate user behavior from start to finish.

- **Set Up Integration Tests**:

  Use the integration_test package to write E2E tests.

  Example:

```
testWidgets('E2E Test Example', (WidgetTester tester) async {
  await tester.pumpWidget(MyApp());
  // Simulate user interactions
});
```

10. Edge Case Tests

- **Identify Edge Cases**:

  Consider unusual input or state scenarios.

- **Write Tests**:

```
test('handles edge case', () {
  expect(() => someFunction(-1), throwsA(isA<ArgumentError>()));
});
```

11. Stress Testing

Stress testing evaluates the application's performance under extreme conditions.

- **Simulate Load**:

Use tools like Apache JMeter or Gatling to simulate high user load.

- **Monitor Performance**:

Check how the app handles the load and identify bottlenecks.

12. Performance Tests

- **Measure Performance**:

Use the flutter_driver or integration tests to measure UI performance.

- **Write Performance Tests**:

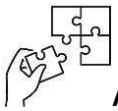
```
testWidgets('performance test example', (WidgetTester tester) async {
  final stopwatch = Stopwatch()..start();
  await tester.pumpWidget(MyApp());
  stopwatch.stop();
  print('Time taken: ${stopwatch.elapsedMilliseconds} ms');
});
```

Points to Remember

- **Testing:** is the process of evaluating a system or its components to determine whether it satisfies the specified requirements and functions correctly.
- **Testing Levels:** Familiarity with different levels of testing (e.g., unit, integration, system, acceptance) and their purposes.
- **There are several Types of Testing such as**: Unit Tests, Widget Tests, Integration Tests, Functional Tests, UI Tests, Performance Tests, Regression Tests, Cross-Platform Testing, Security Testing, End-to-End (E2E) Tests, Mocking and Stubbing, and Code Coverage Tests.
- **Test Reliability such as**: Unit and Widget Testing, Integration and End-to-End Testing, Edge Case and Stress Testing, Performance Testing and User Acceptance Testing (UAT).

- **While implementing the testing pass through the following steps:**

- ✓ Perform Unit tests
- ✓ Perform Widget tests
- ✓ Implement Integration tests
- ✓ Select Testing Tools
- ✓ Perform Test using Emulator and Simulator
- ✓ Perform Test using Physical Device
- ✓ Perform Manual Testing
- ✓ Perform Automated Testing
- ✓ Perform E2E tests
- ✓ Perform Edge case tests
- ✓ Perform Stress testing
- ✓ Execute Performance tests



Application of learning 3.5.

HealthTrack Solutions is preparing to launch a new mobile application called **HealthMate**. This app allows users to track their fitness activities, monitor health metrics, and receive personalized health insights. As the launch date nears, the QA team must ensure the app is reliable and performs optimally. The task is the creating a comprehensive testing strategy including : Performing Unit Tests, Widget Tests, Integration Tests, Testing Using Physical Devices, Manual Testing, Automated Testing, End-to-End (E2E), Edge Case Stress Testing, Selecting Testing Tools, Executing Performance Tests that ensures **HealthMate** is not only functional but also user-friendly and reliable at launch. Each type of testing is crucial for delivering a high-quality application that meets user needs.

Indicative content 3.6 Debug Codebase issues

Duration: 2 hrs

Theoretical Activity 3.6.1: Description of Codebase ssues

Tasks:

1:You are requested to answer the following questions related to the description ofCodebase issues:

i. Define the following terms:
   a. Codebase
   b. Debug
   c. Logging

ii. Differentiate the following terms used in Flutter DevTools:
   a. Isolation
   b. Assertion
   c. Breakpoints
   d. Print Statement

iii. List examples of Applying Debugging Methods

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class.

4: For more clarification, read the key readings 3.6.1. In addition, ask questions where necessary.

Key readings 3.6.1: Description of Codebase issues

✓ **Description of Key Terms**

- **Codebase:**

  Definition: The entire collection of source code for a particular software project, including the files and directories that make up the program.

  Example: A mobile app's codebase may include files for user interface components, business logic, and configuration settings, all organized in a directory structure.

- **Debug:**

  Definition: The process of identifying, isolating, and fixing problems or bugs in the code.

  Example: A developer notices that the app crashes on startup and uses debugging tools to trace the error back to a null pointer exception in the initialization code.

- **Logging:**
  Definition: The practice of recording messages or data about program execution to help track the flow of the application and identify issues.
  Example: Using a logging library to record user actions and error messages, like logger.error("Failed to load user data"), which can help identify issues during runtime.
- **Flutter DevTools:**
  Definition: A suite of performance and debugging tools for Flutter applications that provides insights into app performance, rendering, and state management.
  Example: A developer uses Flutter DevTools to inspect widget trees and identify performance bottlenecks in the app's UI rendering.
  Isolation:
  Definition: The concept of separating code execution environments to prevent one part of the application from affecting another, especially in concurrent programming.
  Example: In Flutter, isolating heavy computations in a separate isolate to keep the UI responsive while processing data.
- **Assertion:**
  Definition: A statement used during development to verify that a condition is true; if the condition is false, the program will throw an error.
  Example: Using assert(x > 0) to ensure that the variable x is always positive during execution, helping catch logic errors early.
- **Breakpoints:**
  Definition: Specific points in the code where execution can be paused to inspect the current state, variables, and execution flow.
  Example: A developer sets a breakpoint on a line of code in their IDE to pause execution when a function is called, allowing them to inspect variable values at that moment.
  Print Statement:
  Definition: A simple debugging technique that involves displaying variable values or messages in the console to trace program execution.
  Example: Inserting print("User input: $userInput") in the code to output the user's input to the console for verification during development.

✓ **Applying Debugging Methods**

Utilizing various techniques and tools to identify and fix issues in the codebase, such as:

- **Using Breakpoints:** Setting breakpoints in the IDE to pause execution.
- **Step Through Code:** Executing code line by line to observe behavior.
- **Analyzing Logs:** Reviewing log files for errors or unexpected behavior.

- **Using Flutter DevTools:** Inspecting the widget tree, performance, and network requests to identify issues.

✓ **Code Reviews**

Definition: A systematic examination of code changes by peers to ensure quality, identify bugs, and improve code structure.

Example: A developer submits a pull request, and team members review the changes, providing feedback on code clarity, potential bugs, and adherence to coding standards.

✓ **Prepare Documentation**

Definition: Creating clear and detailed documentation that explains the codebase, debugging processes, and common issues encountered.

Example: Writing a README file that describes how to set up the development environment, run tests, and troubleshoot common errors, such as "If you encounter a NetworkError, check your API endpoint in the configuration file."

Practical Activity 3.6.2: Debug Codebase issues

Task:

1:Read Key reading 3.6.2

2:Referring to the key reading 3.6.2, You are requested to go to the computer lab to Debug Codebase issues

3:Present your work to the trainer/ your colleagues.

4:Ask for clarification if any.

Key readings 3.6.2: Debug Codebase issues

1. Debugging Tools

Flutter provides several built-in tools for debugging:

Step 1: Use Flutter DevTools

- **Launch DevTools**: Start your Flutter app in debug mode:

```
flutter run --debug
```

- **Open DevTools**: In your terminal, you'll see a URL to open DevTools in your browser. Click on it or navigate to it.

- **Explore Features**:

- **Performance**: Monitor frame rendering and CPU usage.
- **Memory**: Track memory allocation and leaks.
- **Inspector**: Inspect the widget tree and layout issues.

2. Logging

Logging helps track application behavior and diagnose issues.

Step 1: Implement Logging

- **Add Logging**: Use the logger package for structured logging. Add it to your pubspec.yaml:

```
dependencies:
  logger: ^1.0.0
```

- **Initialize Logger**:

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

final logger = Logger();

void myFunction() {
  logger.d('Debug message');
  logger.i('Informational message');
  logger.w('Warning message');
  logger.e('Error message');
}
```

- **View Logs**: Check logs in the console when running your app.

3. Setting Breakpoints

Breakpoints allow you to pause execution and inspect variables.

Step 1: Set Breakpoints

- **Open Your IDE**: Use an IDE like Visual Studio Code or Android Studio.
- **Open Your Code**: Navigate to the file where you want to set a breakpoint.
- **Set Breakpoint**:

- In **VS Code**: Click in the gutter next to the line number.

- In **Android Studio**: Click in the left margin next to the line number.

- **Run in Debug Mode**: Start your application in debug mode. Execution will pause at the breakpoints, allowing you to inspect variable values.

4. Isolation

Isolating parts of your code helps identify issues without interference.

Step 1: Use Isolates

- **Create an Isolate**:

```
import 'dart:isolate';

void isolateFunction(SendPort sendPort) {
  // Perform heavy computations
  sendPort.send('Result from Isolate');
}

void startIsolate() async {
  final receivePort = ReceivePort();
  await Isolate.spawn(isolateFunction, receivePort.sendPort);
  receivePort.listen((data) {
    print('Received: $data');
  });
}
```

- **Identify Issues**: Use isolates for long-running tasks to keep the main thread responsive.

5. Assertions

Assertions help catch errors during development.

- **Add Assertions**: Use assertions to validate conditions,

```
void calculate(int value) {
  assert(value >= 0, 'Value must be non-negative');
  // Proceed with calculation
}
```

- **Run in Debug Mode**: Assertions are only active in debug mode, helping you catch errors without affecting release builds.

6. Code Reviews

Code reviews help identify issues early in the development process.

Step 1: Conduct Code Reviews

- **Set Up a Code Review Process**: Use tools like GitHub, GitLab, or Bitbucket to facilitate code reviews.
- **Review Code Together**: Encourage team members to review each other's code for best practices and potential bugs.
- **Provide Feedback**: Offer constructive feedback on code quality, performance, and readability.

7. Preparing Documentation

Documentation can help clarify the purpose and usage of code.

Step 1: Write Documentation

- **Use Dart Documentation Comments**:

```dart
/// Calculates the sum of two integers.
///
/// Throws an [ArgumentError] if either value is negative.
int add(int a, int b) {
  assert(a >= 0 && b >= 0);
  return a + b;
}
```

- **Create a README**: Write a README file for your project, including:

  - Project overview
  - Installation instructions
  - Usage examples
  - Contribution guidelines

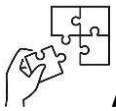
- **Maintain Up-to-Date Documentation**: Regularly update documentation to reflect code changes.

Points to Remember

- **Codebase:** Understanding what a codebase is and its role in software development.
- **Debug:** Debugging in Flutter involves identifying and fixing errors or issues in your code.
- **Logging:** Logging refers to the practice of recording information about application execution to help diagnose issues or understand application behavior.
- **Isolation:** In Flutter, an isolation is a separate memory space where Dart code runs.
- **Assertion:** An assertion is a statement used in Flutter (and Dart) to verify that a condition is true during runtime.
- **Breakpoints:** Breakpoints are markers set in the code that pause execution at a specific line during debugging.
- **Print Statement:** A print statement in Flutter is a simple way to output information to the console for debugging purposes.
- **Flutter DevTools:** Familiarity with Flutter DevTools and its features for debugging Flutter applications
- **Applying Debugging Methods:** Theoretical understanding of various debugging methods and strategies, including systematic approaches to isolate and fix issues.
- **Breakpoints:** Understanding how breakpoints work in debugging, allowing developers to pause execution at specific points**.**

✓ **While Debugging Codebase issuespass through the following steps:**

- Use Debugging Tools
- Implement Logging
- Perform Setting Breakpoints
- Use Isolation is used
- Use Assertions
- Perform Code Reviews
- Perform Preparing Documentation



Application of learning 3.6.

EduTech Corp is developing a new online learning management system (LMS) called **LearnHub**. The development team faces challenges related to user progress tracking and content delivery, which are critical for a seamless user experience. The task is the integrations various practices of debugging tools, logging, breakpoints, assertions, code reviews, and documentation into their workflow. By addressing these questions, they can enhance the robustness and maintainability of the **LearnHub** platform, ultimately leading to a better user experience.

Learning outcome 3 end assessment

Theoritical assessment

I. **Circle the letter corresponding to the correct answer:**

1. Which HTTP method is primarily used to create a new resource on the server?

A) GET

B) POST

C) PUT

D) DELETE

2. What is the primary purpose of the PUT method in RESTful APIs?

A) Retrieve data

B) Update an existing resource

C) Delete a resource

D) Create a new resource

3. Which of the following is essential for securely handling user authentication in a mobile application?

A) Using only GET requests

B) Storing passwords in plain text

C) Utilizing HTTPS for API calls

D) Ignoring token expiration

4. What is the purpose of the PATCH method in RESTful services?

A) Delete a resource

B) Update part of a resource

c) Retrieve a resource

II. **Answer by True to the correct statements or False to the wrong statements**

1. Data integrity refers to the protection of data from unauthorized access and modifications……..
2. Using encryption is a common practice to enhance security standards for data stored in a database……..

3. Regular audits and monitoring are not necessary for maintaining data integrity in a backend system……..

4. Security standards such as GDPR and HIPAA help organizations ensure the confidentiality and integrity of sensitive data……

5. Implementing strong access control measures can help maintain both data integrity and security in a backend application……..

II. **Fill the empty space by using the correct word listed in the bracket.**

1. In software development, a _____ is the collection of source code used to build a software program. (Logging, Breakpoints, Codebase, Assertion, Debug, Isolation)

2. The process of identifying and fixing bugs in your application is known as _____. (Logging, Breakpoints, Codebase, Assertion, Debug, Isolation)

3. _____ is the practice of recording information about the execution of a program to help diagnose issues. (Logging, Breakpoints, Codebase, Assertion, Debug, Isolation)

4. In testing, _____ refers to the practice of running tests in a way that ensures they do not affect each other. (Logging, Breakpoints, Codebase, Assertion, Debug, Isolation

5. A(n) _____ is a statement used in programming that checks if a condition is true, often used in testing. (Logging, Breakpoints, Codebase, Assertion, Debug, Isolation)

6. Developers use _____ to pause the execution of a program at a specific line of code to examine the state of the application.

III. Match the following terms to their descriptions and write letters in an empty place of answers.

Answers	**Terms**	**Descriptions**
1…**……**	1) Unit Tests	A) 1. Tests the performance and responsiveness of the application
2…..**..**	2) Widget Tests	B) Tests the interaction between different modules of the application.
3……**..**	3) Integration Tests	C) Tests individual components or functions in isolation.
4………	4) Functional Tests	D) Tests the application against functional requirements.
5……	5) UI Tests	E) Tests the user interface for usability and visual correctness
6…….	6) Performance Tests	F) Ensures that previously developed and tested software still works after a change.
		G) Focuses on testing performance under various conditions

Practical assessment

At TechSolutions Inc, the development team is tasked with creating a mobile application that integrates various backend functionalities to enhance user experience. The application requires efficient handling of user data through various HTTP requests: POST,GET, PUT, DELETE and PATCH. The application is designed using the implementation of Microapps allows for easier maintenance and scalability, the debugging of the Codebase Issues , the performing of Error Handling and the performing of Testing.

References

https://www.geeksforgeeks.org/what-is-ide/

https://www.w3schools.com/

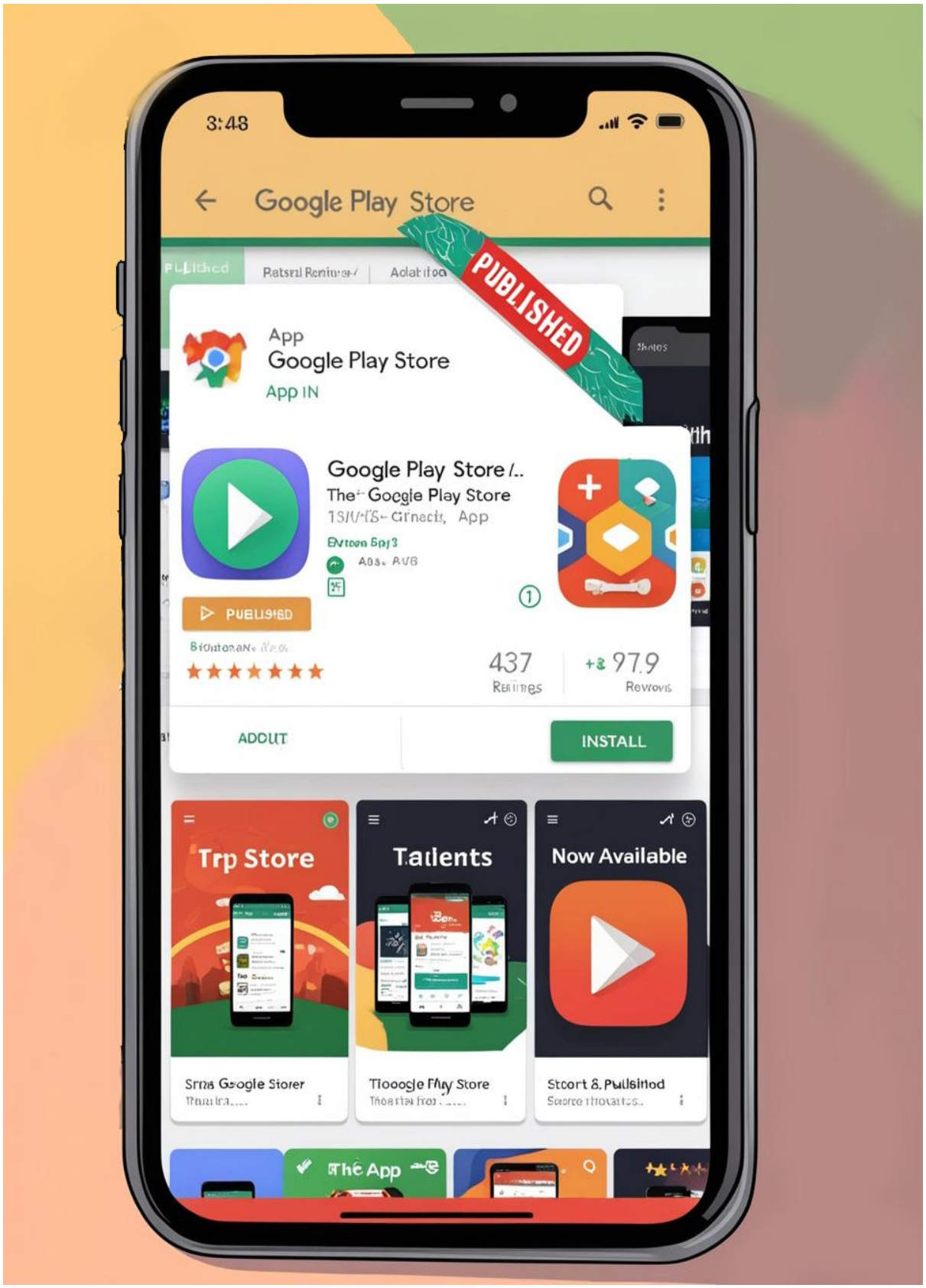
https://stackoverflow.com/

https://chatgpt.com/g/g-RGr8YRENd-chart-gpt-3

Napoli, M. L. (2019). Beginning Flutter: A Hands On Guide to App Development 1st Edition. Indianapolis: John Wiley & Sons.

Rose, R. (2022). Flutter and Dart Cookbook 1st Edition. Calfornia: O'Reilly Media.

Miola, A. (2020). Flutter Complete Reference: Create Beautiful, Fast and Native Apps for Any Device. KDP: Amazon LLC.


3:48
Google Play Store
PUBLISHED
App
Google Play Store
App IN
INSTALL
Now Available

Indicative Contents
4.1 Generation of Installable Files
4.2 Submission of Application Files
4.3 Address Post Deployment Issues

Key Competencies for Learning Outcome 4: Publish Application

Knowledge	Skills	Attitudes
● Description of Installable files for mobile platform ● Description of application files submission ● Description Addressing post deployment issues	● Generating installable files for mobile platform ● Performing applications submission ● Applying the address post deployment issues.	● Having Team work ability ● Being critical thinker ● Having Passion for Learning ● Having Problem-Solving Mindset ● Having good Collaboration and Communication ● Being Attentive to Security

Duration: 20 hrs

Learning outcome 4 objectives:

By the end of the learning outcome, the trainees will be able to:

1. Describe correctly installable files for mobile platforms.
2. Generate correctly installable filles for mobile application.
3. Describe clearly the application files as used in mobile application.
4. Perform properly the application files as applied in mobile application.
5. Describe appropriately the address post deployment issues as used in mobile application.
6. Apply the address post deployment issues as used in mobile application.

Resources

Equipment	Tools	Materials
● Computer ● Mobile phone	● Flutter Inspector ● DevTools ● Browser	● Internet ● Electricity

Indicative content 4.1: Generation of installable files

Duration: 10 hrs

Theoretical Activity 4.1.1: Description of Generation of installable files

Tasks:

1: You are requested to answer the following questions:

1. What are the differences between a Debug build and a Release build?
2. When would you typically use a Debug build versus a Release build?
3. What is the format of installable files for Android and iOS?
4. What is the purpose of an.APK file?
5. What does Just-in-Time (JIT) compilation do?
6. Is JIT compilation done at runtime or before the app runs?
7. What is Ahead-of-Time (AOT) compilation?
8. What are the advantages of Ahead-of-Time (AOT) compilation for performance?
9. What is the difference between Hot Reload and Hot Restart?
10. What tool do you use to generate an **.IPA** file for iOS?

2: Provide the answer for the asked questions and write them on papers.

3: Present the findings/answers to the whole class

4: For more clarification, read the key readings 4.1.1. In addition, ask questions where necessary.

Key readings 4.1.1: Description of Generation installable files

1. **Installable files**

   Installable files in mobile applications are the packages that contain all the necessary components required to install and run an app on a mobile device. These files vary depending on the platform (Android or iOS) and the method of distribution. Here's a clear breakdown of the types of installable files used in mobile applications:

- **For Android Installable Files**
  - **APK (Android Package Kit):**The standard file format for installing applications on Android devices. An APK file contains all the app's code, resources, assets, and manifest file. Users can install APKs directly on their devices or through the Google Play Store.

- **AAB (Android App Bundle):**A publishing format that includes all the app's compiled code and resources but does not generate APKs until the app is downloaded from the Play Store. AAB allows for optimized delivery, meaning users only download the parts of the app that are relevant to their device configuration.
- **XAPK:**This format combines an APK file with additional data files (like OBB files) that are often needed for larger applications, especially games. XAPK files are typically used when the app's size exceeds the limits of a standard APK.

For iOS Installable Files

- **IPA (iOS App Store Package):**The file format used for iOS applications. An IPA file contains the app binary and is used for distribution through the App Store or for enterprise distribution. Users typically install IPA files through the App Store or via Xcode during development.
- **DEB:**A package format used for jailbroken iOS devices, allowing users to install apps that are not available on the App Store. DEB files are not commonly used for standard app distribution.

These installable files are crucial for deploying mobile applications effectively on their respective platforms.

2. Compilation Models

In mobile application development, **Compilation Models** refer to the different approaches used to compile and run mobile apps on devices. There are three primary compilation models for mobile apps:

I. Just-in-Time (JIT) Compilation

JIT compilation compiles parts of the code at runtime, right before it is executed. Instead of compiling the entire application beforehand, JIT dynamically converts the app's bytecode into native machine code during execution. Android apps often use JIT with the Android Runtime (ART).

Advantages:

- **Faster initial build times**: Since code is not fully compiled before being run, the development build process is faster, leading to quicker iterations during development.
- **Memory efficiency**: Only the parts of the app that are being executed are compiled, reducing the app's memory footprint compared to Ahead-of-Time compilation.

Disadvantages:

- **Slower runtime performance**: Since compilation occurs while the app is running, this can slow down app performance compared to fully compiled Ahead-of-Time apps.
- **Higher battery usage**: The on-the-fly compilation can increase CPU usage, which leads to higher battery consumption on mobile devices.

II. Ahead-of-Time (AOT) Compilation

In AOT compilation, the entire code is compiled into native machine code before the application is deployed or run on the device. This results in better performance since the app runs directly as native code. Frameworks like Flutter use AOT to ensure smooth performance.

Advantages:

- **Better runtime performance**: Since the code is precompiled into native machine code, the app runs more efficiently, with faster execution times and smoother performance.
- **Faster startup times**: Because the app does not need to compile any code during execution, it loads more quickly.

Disadvantages:

- **Longer build times**: Compiling all the code ahead of time can result in longer build times during development, which may slow down iterations when making changes to the app.
- **Larger app size**: The fully compiled app may take up more storage space on the device because all of the app's code is precompiled.

III. Hot Reload and Hot Restart mechanisms

Hot Reload and Hot Restart are mechanisms used to speed up the development process by allowing developers to see changes in their code without needing to restart the entire app. These mechanisms are typically used in frameworks like **Flutter** and **React Native**.

- **Hot Reload** injects updated source code into a running application and applies changes instantly without losing the app state.
- **Hot Restart** reloads the entire app from scratch, which resets the app's state but allows for full application updates.

Advantages:

- **Faster iteration cycles**: With Hot Reload, developers can instantly see changes in the UI and app logic without restarting, allowing for more rapid debugging and testing.
- **Preserving app state**: Hot Reload keeps the app's state intact, so developers can modify the code without having to restart the app or navigate back to a specific screen.

Disadvantages:

- **Partial updates**: Hot Reload may not reflect changes that require a full recompile or affect app initialization, requiring a Hot Restart or a full rebuild.
- **Occasional inconsistencies**: In some cases, Hot Reload can cause inconsistencies or crashes, especially when there are significant changes in the codebase that require a full restart to work correctly.

3. Builds generation

To generate mobile app builds for **iOS** and **Android**, developers create two specific file types: **IPA** (for iOS) and **APK** (for Android). Here's a simplified explanation of how this process works for each platform:

✓ iOS (IPA Generation)

An **IPA** (iOS App Store Package) file is the format used to package iOS apps so they can be installed on iPhones or iPads. The process for generating an IPA involves:

- **Development Environment**: You use **Xcode**, which is Apple's official development tool.
- **Code Signing**: iOS apps must be signed with a valid certificate and provisioning profile (provided by Apple) to be installed on a real device or uploaded to the App Store.
- **Build Process**: In Xcode, you compile your app's code, assets, and resources into a single IPA file.
- **Deployment**: Once the IPA is generated, you can either:
  - Upload it to the **App Store** for public distribution.
  - Distribute it directly to devices using **TestFlight** or other internal methods for testing.

✓ **Android (APK Generation)**

An **APK** (Android Package Kit) is the file format used for distributing and installing

apps on Android devices. To create an APK:

- **Development Environment**: You use **Android Studio**, Google's official development tool for Android apps.
- **Build Variants**: You can generate different types of APKs, such as:
  - **Debug APK**: Used for testing and debugging on devices.
  - **Release APK**: A signed version for distribution on Google Play or other app stores.
- **Build Process**: In Android Studio, your app's code, assets, and resources are compiled into a single APK file.
- **Deployment**: Once the APK is built, you can:
  - Upload it to the **Google Play Store** for distribution.
  - Share it directly with users for testing or local installation.

Practical Activity 4.1.2: Generating Installable files

Task:

1: Read key reading 4.1.2

2: Referring to the key reading 4.1.2**,**You are requested to go to the computer lab to Generate Installable a Debug**.APK** file in Android studio.

3: Present your work to the trainer and whole class.

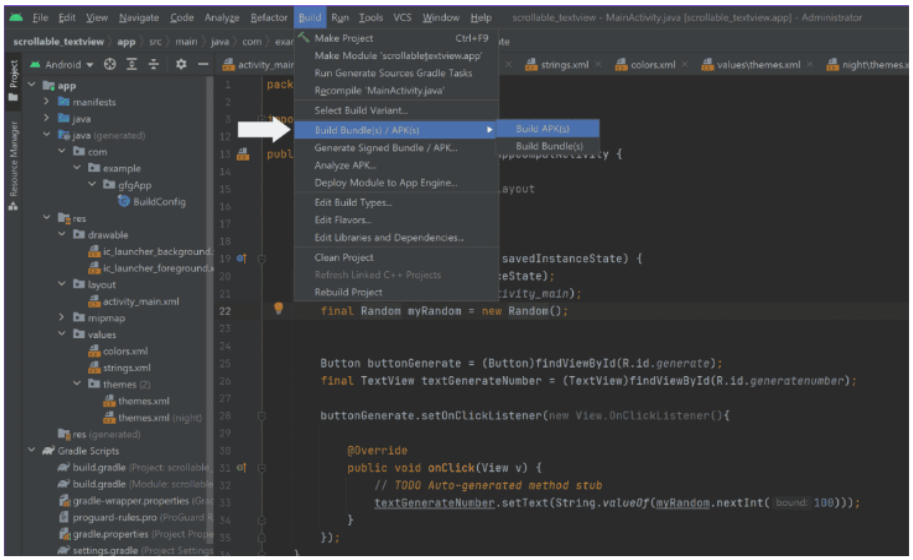
4: Ask for clarification if any**.**

Key readings 4.1.2: Generating Installable files

Generate an Installable a Debug. APK File in Android Studio:

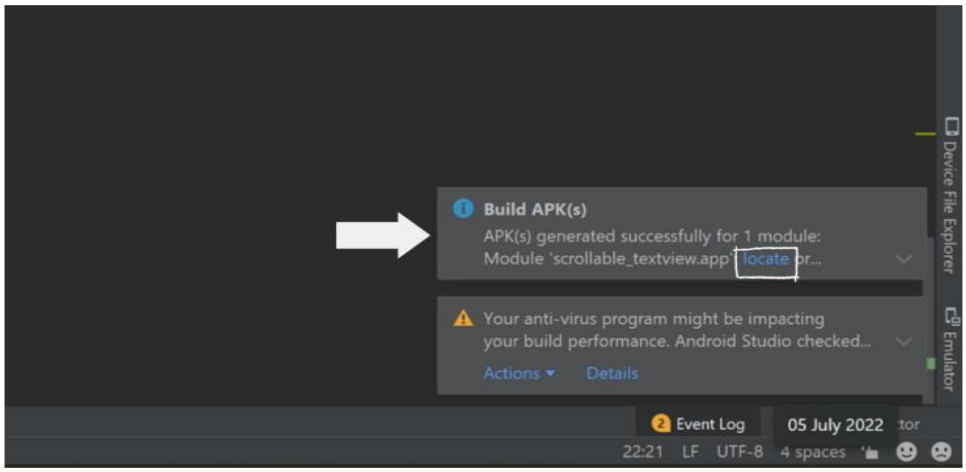
Steps:

1. After building a project/application that you want to import into an APK file:
2. Open Android Studio:Launch Android Studio and open your project.
3. Go to Build > Build Bundle(s)/APK(s) > Build APK(s) from the toolbar menu.



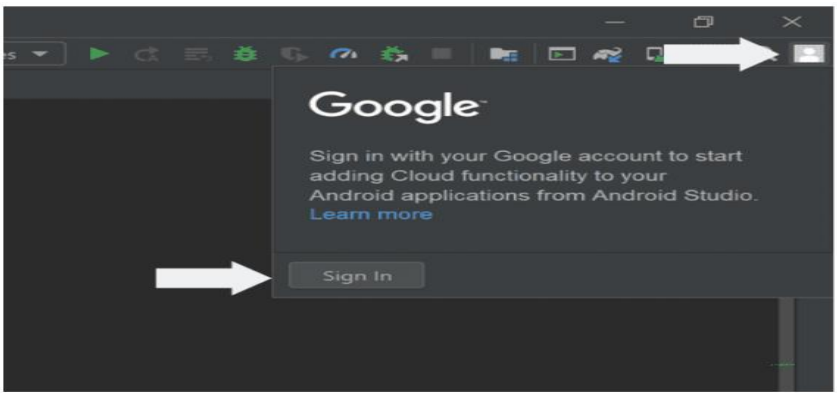
Output:

Android Studio will take a few minutes to generate the APK file. Once the APK file build is complete, you'll receive a notification on the bottom right corner of your screen. Select Locate and you will find the APK file location. If you miss the notification, you can still find the APK file within the project folder: app/build/outputs/apk/debug. The file by default is named app-debug.apk.

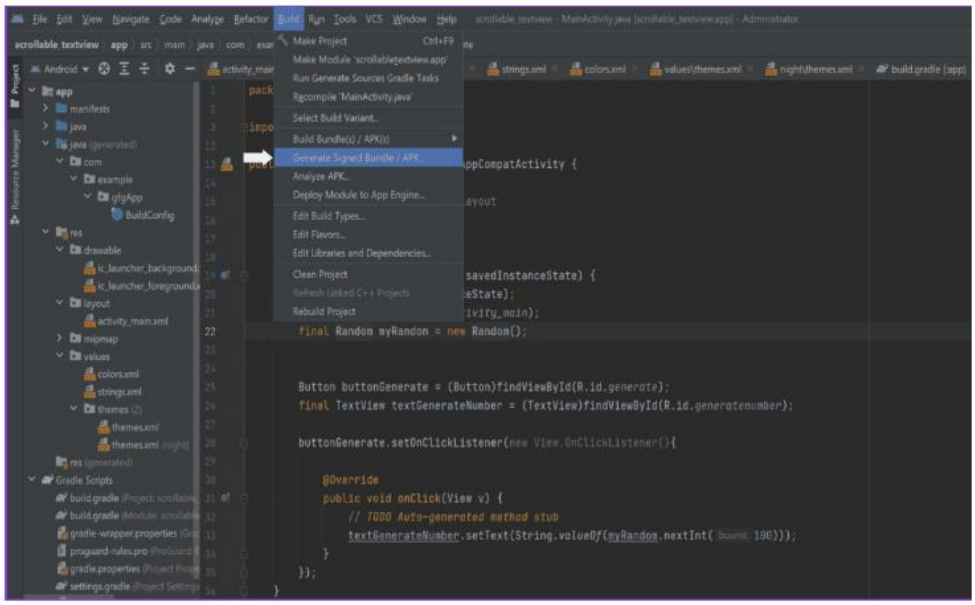


Generate an Installable a Signed. APK File in Android Studio:

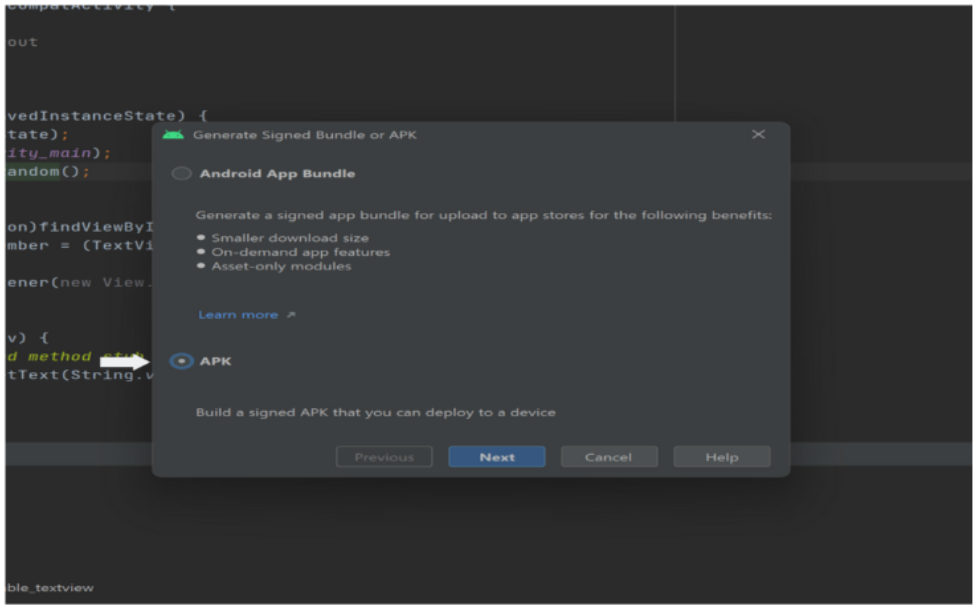
Step 1: Sign in to your Google Account as shown below. If you are already logged in move to the next step.



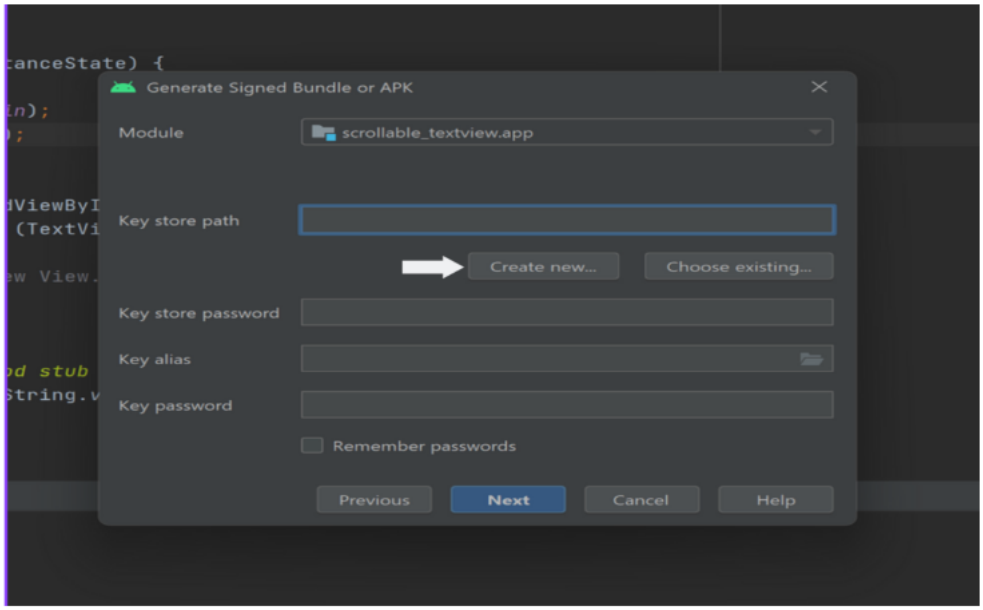
Step 2: From the toolbar, click on the 'Build' option and select the 'Generate Signed Bundle / APK as shown in the image below.



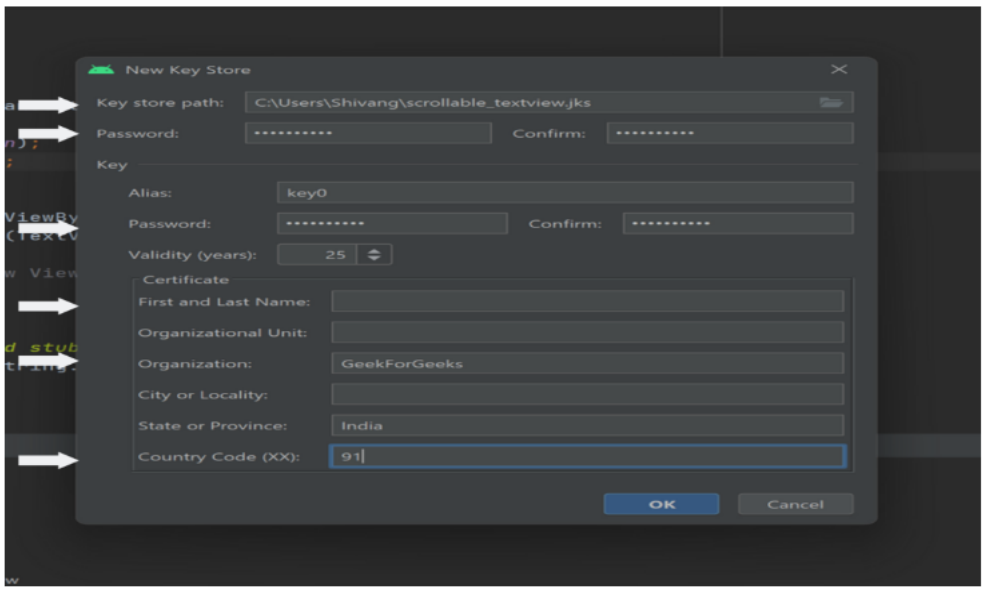
Step 3: Now on the appeared pop-up window select ‘APK’ in the radio button as shown in the image below. Then, click on the ‘Next’ button.



Step 4: Now Create a new KeyStore credential for your App by clicking on “Create New”.



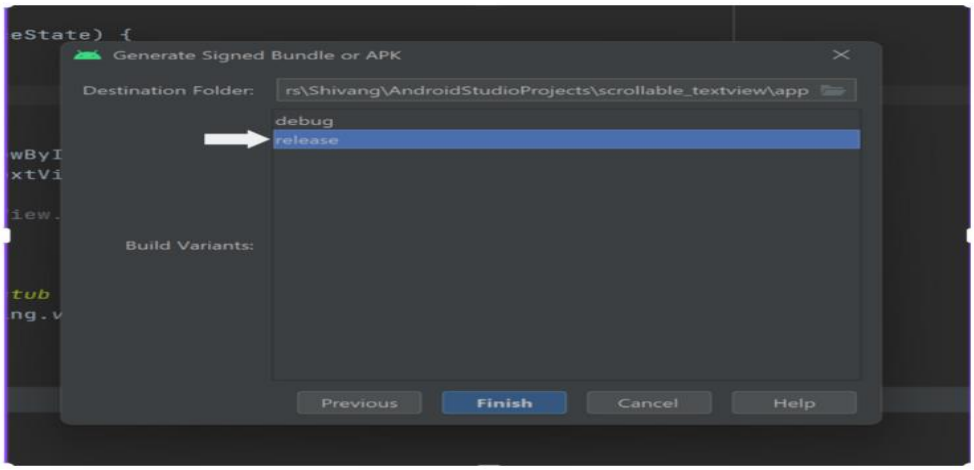
Step 5: In 'New Key Store', browse a 'Key store path' by clicking on the Folder icon as shown. Then, provide a 'Password' and 'Confirm' password. Now, on the 'Key' section, provide an 'Alias:' or go with the already provided one. After that enter a 'Password' and 'Confirm' password. After that, you can move on to the 'Certificate' section. here, provide your 'First and Last Name'. Enter your address details. After you have provided the required details, click on the 'OK'.



Step 6: You will see the path for your KeyStore with the password you gave. Click on next.

Step 7: On the pop-up window select Release and end the process by clicking on

Finish as shown below.



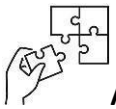
Output: Android Studio will take a few minutes to generate the APK file. Once the APK file build is complete, you'll receive a notification on the bottom right corner of your screen. Select Locate and you will find the APK file location. The Signed APK file is by default named app-release.apk. You will find it in the project folder in the app/release directory.

Points to Remember

- **Android Installable Files:**
  - ✓ **APK:** Standard Android app file, installable directly or via Google Play.
  - ✓ **AAB:** Optimized delivery format used for Google Play.
  - ✓ **XAPK:** Combines APK with extra data for larger apps.
- **iOS Installable Files:**
  - ✓ **IPA:** Standard file for iOS apps, distributed via App Store or TestFlight.
- **Compilation Models:**
  - ✓ **Just-in-Time (JIT) Compilation:** Compiles during runtime. Fast builds, slower performance.
  - ✓ **Ahead-of-Time (AOT)Compilation**: Pre-compiles code. Better performance, longer build times.
  - ✓ **Hot Reload & Hot Restart:** Quick code updates. Fast iteration, potential inconsistencies.

- **Builds Generation:**
    - ✓ **iOS (IPA):** Use Xcode, sign with certificates, deploy via App Store or TestFlight.
    - ✓ **Android (APK):** Use Android Studio, generate APKs for testing or release, deploy via Google Play.
- **While Generating Installable files, pass on the following steps:**

  For Android Installable Files
    - ✓ APK (Android Package Kit)
    - ✓ AAB (Android App Bundle)
    - ✓ XAPK



Application of learning 4.1.

XY Company aims to develop a simple calculator application that can perform basic arithmetic operations: addition, subtraction, multiplication, and division. After completing the development of the app, they will generate installable files on Android or iOS platforms.

Indicative content 4.2: Submission of application files

Duration: 5 hrs

Theoretical Activity 4.2.1: Description of Submission of application files

Tasks:

1.You are requested to Answer the following:

i. What are the differences between App Icon and App Screenshot
ii. What are the requirements and implications of having a developer account for app distribution?
iii. What are the build process for mobile applications and the differences between various formats (ipa for iOS, .aab for Android).
iv. What is the significance of App IDs
v. How are Google Developer Account and Apple Developer Account created?
vi. Explain the processes of Upload App bundles (Android & iOS)

2. Provide the answer for asked questions and write them on papers

3.Present the findings to the whole class

4.For more clarification, read the key readings 4.2.1. ask questions where necessary.

Key readings 4.2.1: Description of Submission of application files

Once the installable files (APK for Android, IPA for iOS) are generated, the next step is to **submit the application** to the app stores. This process involves preparing the necessary **store assets** and completing various submission requirements.

Prepare Store Assets

These assets help users understand what the app does and make it visually appealing on the app store.

1. App Icon

✓ **Purpose**: The app icon is the graphical representation of the application on the home screen and in the app store.

✓ **Requirements**:

- Ensure it follows the platform's design guidelines.
- Use high-resolution images: e.g., 1024x1024 pixels for iOS and Android.

- Keep it simple, clean, and identifiable at smaller sizes.

2. App Screenshots

✓ **Purpose**: Screenshots give potential users a visual preview of the app's interface and functionality.
✓ **Requirements**:
- Take screenshots of key features and user interface sections of the app (home screen, calculator functions).
- For Android: Provide screenshots for different screen sizes (phone, tablet).
- For iOS: Supply screenshots for different device types (iPhone, iPad).
- Ensure a minimum number of screenshots (usually 4-5) is provided for each supported device.

3. App Promotional Materials/Previews

✓ **Purpose**: Promotional materials (like banners or video previews) help market the app effectively in the app store.
✓ **Requirements**:
- Create a short video preview (15-30 seconds) demonstrating key features of the app.
- Upload high-quality promotional images or banners (used for App Store or Google Play featured sections).
- Follow platform-specific requirements for file formats and dimensions.

Creating Developer Account Registration

To submit apps to the app stores, you need to create developer accounts on both **Google Play** and **Apple App Store**. Here's a simplified guide for each:

1. Google Developer Console (For Android)

Steps:

1. **Visit the Google Play Console**: Go to the official **Google Play Console** website.
2. **Sign in with Google Account**: Use an existing Google account or create a new one.
3. **Register as a Developer**:

✓ Follow the steps to register as a developer.
✓ Pay the **one-time registration fee** of $25.

4. **Fill in Account Details**: Provide your name, email, and business information.
5. **Accept Developer Agreement**: Review and agree to Google's Developer

Distribution Agreement.

6. **Submit and Wait for Approval**: Once submitted, wait for approval to start publishing apps on the **Google Play Store**.

2. Apple Developer Account (For iOS)

Steps:

1. **Visit Apple Developer Website**: Go to the **Apple Developer** site.
2. **Sign in with Apple ID**: Use your existing Apple ID or create a new one.
3. **Join the Apple Developer Program**:

✓ Click **Enroll** and follow the steps to join.
✓ Pay the **annual fee** of $99.

4. **Complete Enrollment**: Provide necessary details like legal name, address, and business information
5. **Accept the License Agreement**: Review and accept Apple's terms
6. **Wait for Approval**:Once approved, you can publish apps on the **Apple App Store** and use developer tools like **Xcode** and **TestFlight**.

Generate App Release Builds(.ipa, .aab)

1. Generate. IPA File for iOS

Steps:

1. **Open Xcode** and load your iOS project.
2. **Select Generic iOS Device** from the top device menu.
3. **Go to Product > Archive** to build the app.
4. In the **Organizer** window, click **Distribute App**.
5. Choose **App Store Connect** (for App Store) or **Ad Hoc** (for internal testing).
6. Follow the steps and **export the .IPA file**.

2. Generate. AAB File for Android

Steps:

1. **Open Android Studio** and load your project.
2. **Set Build Variant** to **Release** (in the Build Variants panel).
3. Go to **Build > Generate Signed Bundle / APK**.
4. Choose **Android App Bundle (AAB)** and click **Next**.
5. Use or create a **keystore** for app signing.
6. Click **Finish** and the **.AAB file** will be created in app/build/outputs/bundle/release/.

Configure app setting

1. App ID (Android & iOS)

Purpose: The unique identifier for your app, used by app stores to distinguish it from others.

Android: Set this in the build.gradle file (e.g., com.companyname.appname).

iOS: Set the **Bundle Identifier** in **Xcode** (e.g., com.companyname.appname).

2. App Description

Purpose: A clear explanation of what your app does, written to help users understand its purpose and features.

Tip: Keep it concise, highlighting the key features, benefits, and target audience.

3. Set Up App Listing

Purpose: The information users see on the app store (title, description, screenshots, etc.).

Includes:

a. **Title**: App name.

b. **Icon**: High-quality app icon.

c. **Screenshots**: Visual previews of the app.

d. **Categories**: Choose the right category (e.g., education, productivity).

4. Distribution

Purpose: Choosing how and where to release your app.

Android: Upload to **Google Play Store** for global or regional distribution.

iOS: Submit to **Apple App Store** or distribute through **TestFlight** for testing.

Upload App bundles (Android & iOS)

Steps:

1. **Go to App Store Connect**: Visit **App Store Connect** and log in with your Apple Developer account.
2. **Create a New App**:

✓ Click the **My Apps** section, then click the **+** button to add a new app.

✓ Fill in the app name, bundle ID, and other basic details.

3. **Upload the. IPA File**: Open **Xcode** or use the **Transporter app** to upload your . IPA file to App Store Connect.
4. **Prepare App Listing**: Complete your app's listing with description, screenshots, category, and pricing information.
5. **Submit for Review**: After uploading the IPA, review the app details and submit it for review.

Practical Activity 4.2.2: Creating app store assets

Task:

1: Read key reading 4.2.2

2: Referring to the key reading 4.2.2**,** you are requested to go to the computer lab toto Create Google Play and Apple Developer Account.

3: Present your work to the trainer and whole class.

4: Ask for clarification if any

Key readings 4.2.2: Creating app store assets:

- **Creating App Icon in flutter**

Steps:

1. **Prepare Icon Image**

Task: Prepare a square image with dimensions (e.g., 1024x1024 pixels) in **PNG** format.

Goal: This image will be used as the base for your app icon across different platforms.

Add flutter_launcher_icons Package

a. **Task**: Open your **pubspec.yaml** file and add the **flutter_launcher_icons** package under **dev_dependencies.**

dev_dependencies:

flutter_launcher_icons: ^0.9.2

b. **Next**: Set up the configuration for the app icon. Add this section in the same **pubspec.yaml** file:

flutter_icons:

android: true

ios: true image_path: "assets/icon/icon.png"

Replace **"assets/icon/icon.png"** with the path to your icon image.

2. **Run the Flutter Launcher Icons Script**

Task: Open the terminal in your project folder and run the following command:

flutter pub run flutter_launcher_icons: main

Result: This generates the app icon for both Android and iOS in various sizes, automatically adding them to the appropriate directories.

After completing these steps, your app will have the specified icon across platforms.

- **Creating app screenshots**

Steps:

1. **Set Up Emulator or Device:** Run your app on a simulator, emulator, or physical device.
2. **Capture Screenshot: Take a screenshot using the emulator or the device's screenshot functionality:**

✓ For Android Emulator: Press **Ctrl + S** or use the screenshot button in the emulator toolbar.

✓ For iOS Simulator: Press **Cmd + S**.

✓ On a physical device, use the native screenshot gesture.

3.Optional - Use flutter_native_screenshot Plugin

✓ If you prefer taking screenshots programmatically, you can use the **flutter_native_screenshot** package:

1. Add it to your **pubspec. Yaml:**

dependencies:

flutter_native_screenshot: ^1.0.0

*2.*Take a screenshot with this code:

import 'package:flutter_native_screenshot/flutter_native_screenshot.dart';

String path = await FlutterNativeScreenshot.takeScreenshot();

Practical Activity 4.2.3: Creating Google Play or Apple Developer Account

Task:

1: read key reading 4.2.3

2: Referring to the key reading 4.2.3**,**You are requested to go to the computer labto Create Google Play or Apple Developer Account.

3: Present your work to the trainer and whole class.

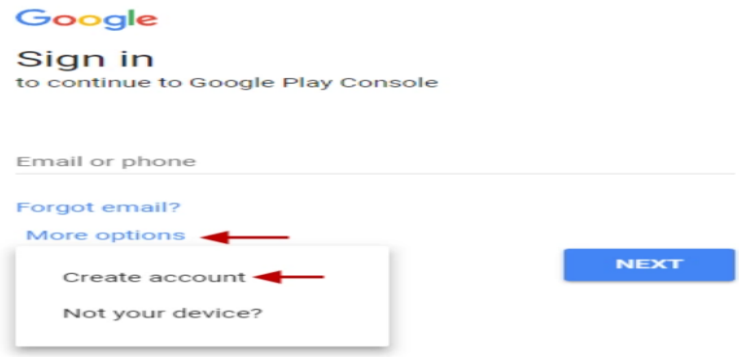
4: Ask for clarification if any

Key readings 4.2.3: Creating Google Play and Apple Developer Account

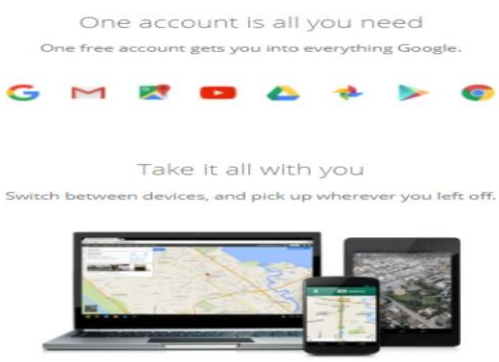
Creating Google Play Developer Account

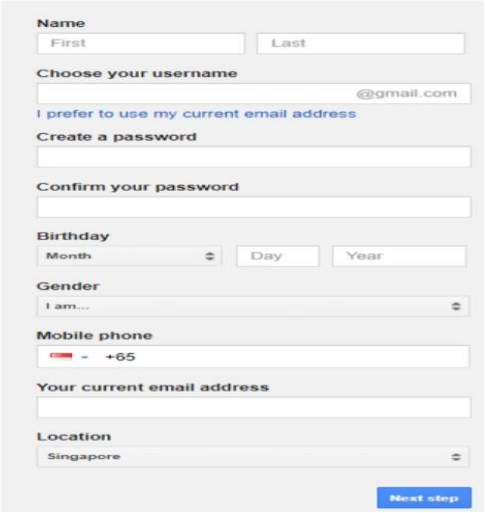
Steps:

1. Go to https://play.google.com/apps/publish/signup/. Click "Create account".

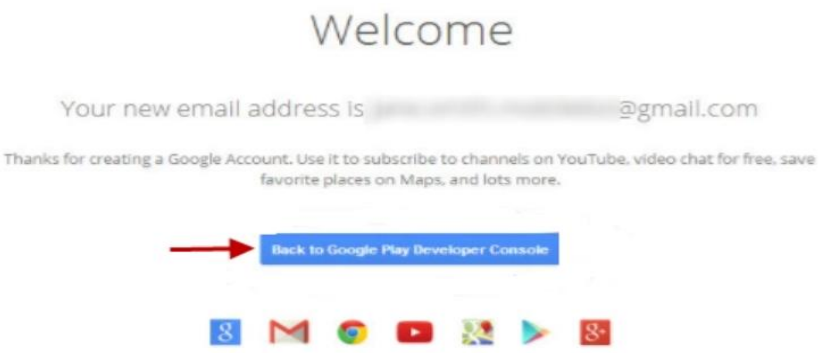


2.Fill out the form, and click "Next Step".(Google Play charges $25 registration fee)

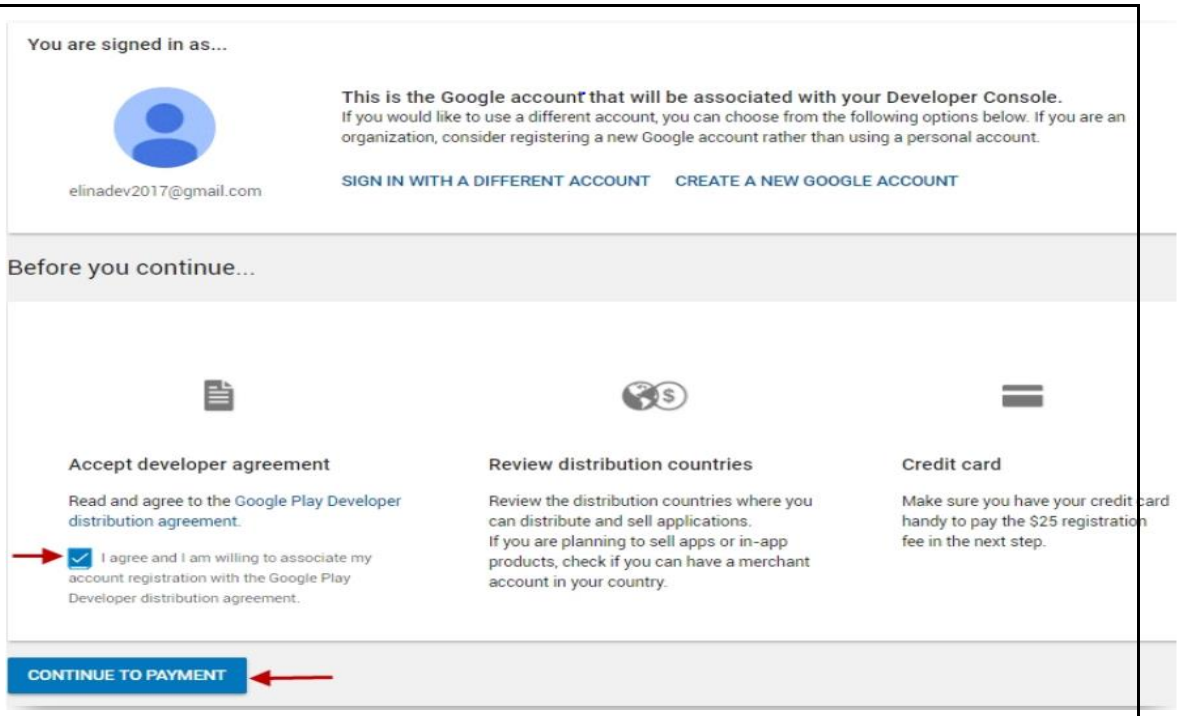




3. Click "Back to Google Play Developer Console".



4. Read and check off that you agree to the Google Play Developer distribution agreement, and click "Continue to Payment" to pay the $25 registration fee.

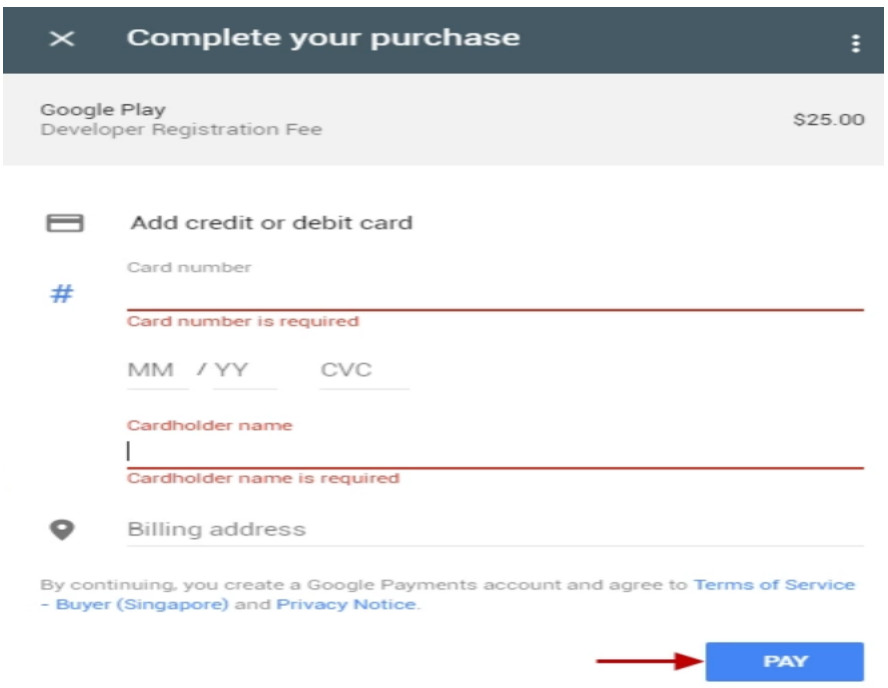


5. Enter your payment information, and click "Pay".

6. You will receive an email from Google notifying you when your account is ready to be activated.This process can take up to a week.

7. Activate your account.

You can now submit your app to the Google Play store.



6. You will receive an email from Google notifying you when your account is ready to be activated.This process can take up to a week.

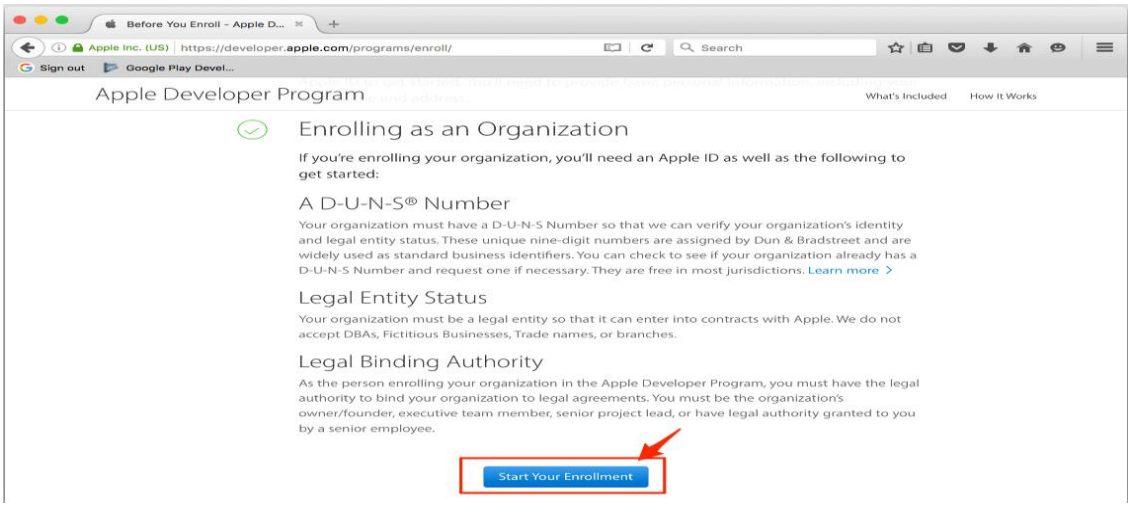
7. Activate your account.
You can now submit your app to the Google Play store.

Creating Apple Developer Account

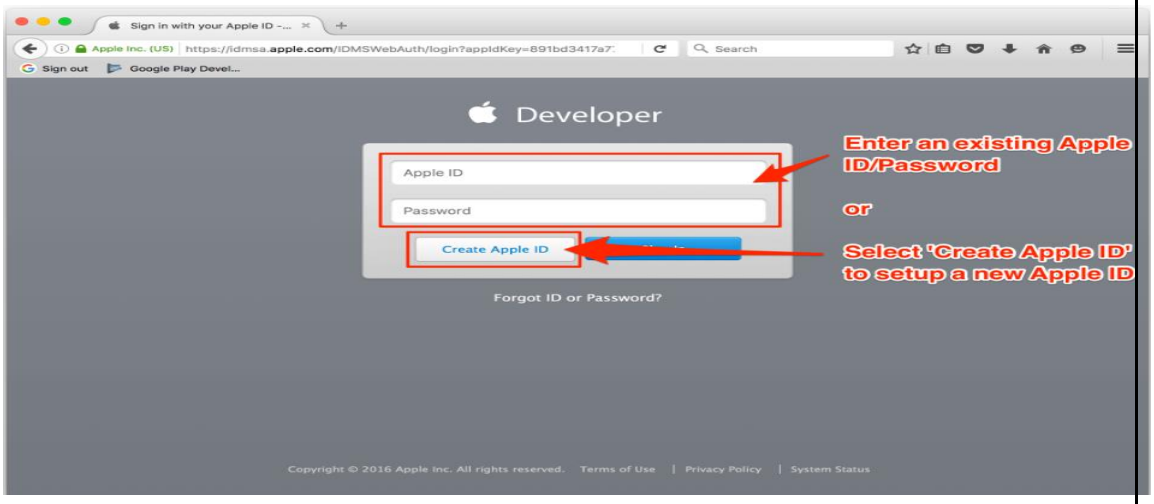
Steps:

1. Visit the URL https://developer.apple.com/enroll/ and click over Start Your Enrollment.

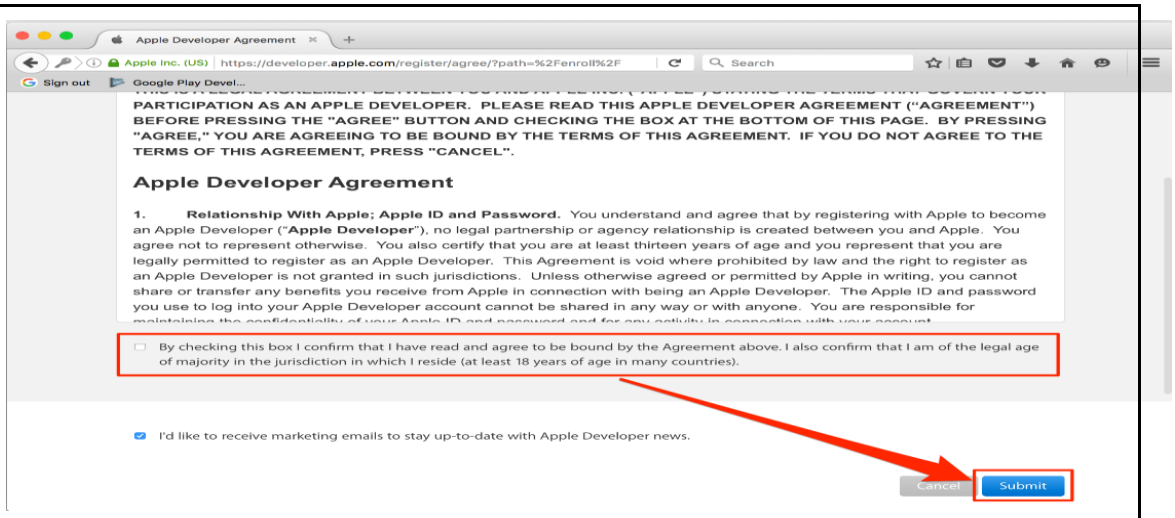
Read more here: https://blog.singsys.com/create-apple-developer-account-in-easy-steps/



2. You can either sign in with an existing Apple ID or create a new Apple ID by clicking 'Create Apple ID'.



3. Now you need to completely read the Apple Developer Agreement before checking the 'By Checking this box I confirm that I have read and agree to be bound by the Agreement above….' box and click the 'Submit' button.



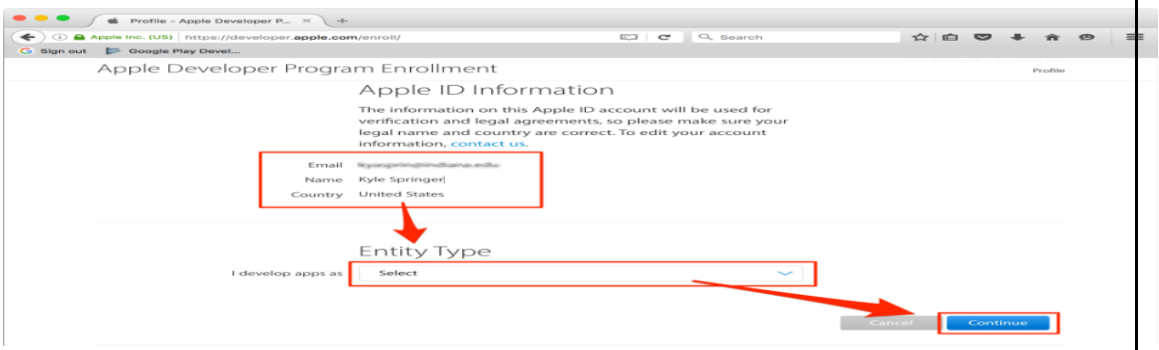
4. In this step you need to confirm that your email address, name, and country location is correct before selecting an appropriate option from the Entity Type dropdown. The Developer name listed on the iTunes App Store is on-behalf of the account type selected from the options below.

Apps published by Individual Developer Accounts will display the name present on the Apple Developer Account.

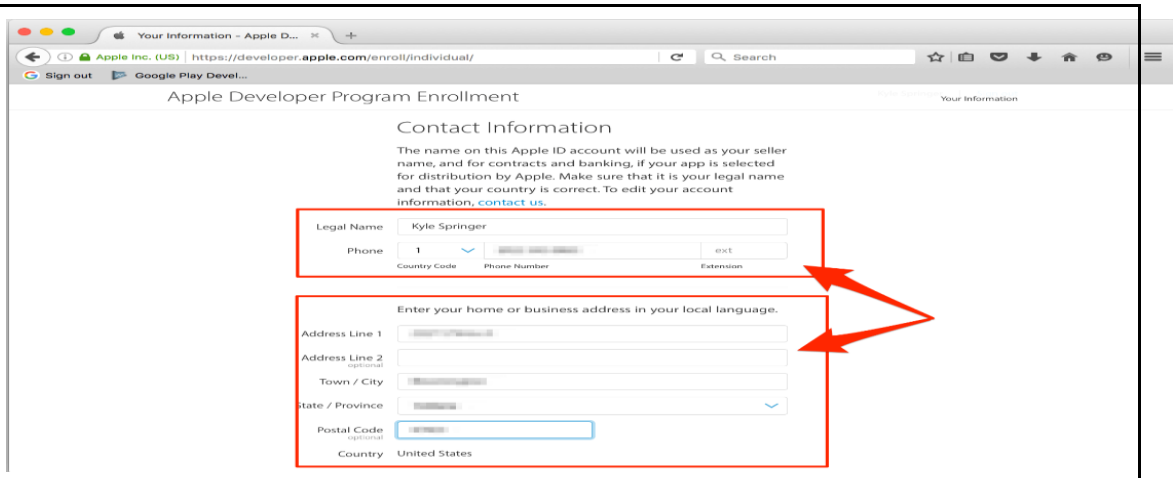
Apps published by Company/Organization Developer Accounts will display the Company Name present in the Apple Developer Account.

Individual/Sole Proprietor/Single Person Business: In case an Individual or a company without an official business can choose this option that will allow creation of single primary login in the Apple Developer Account.

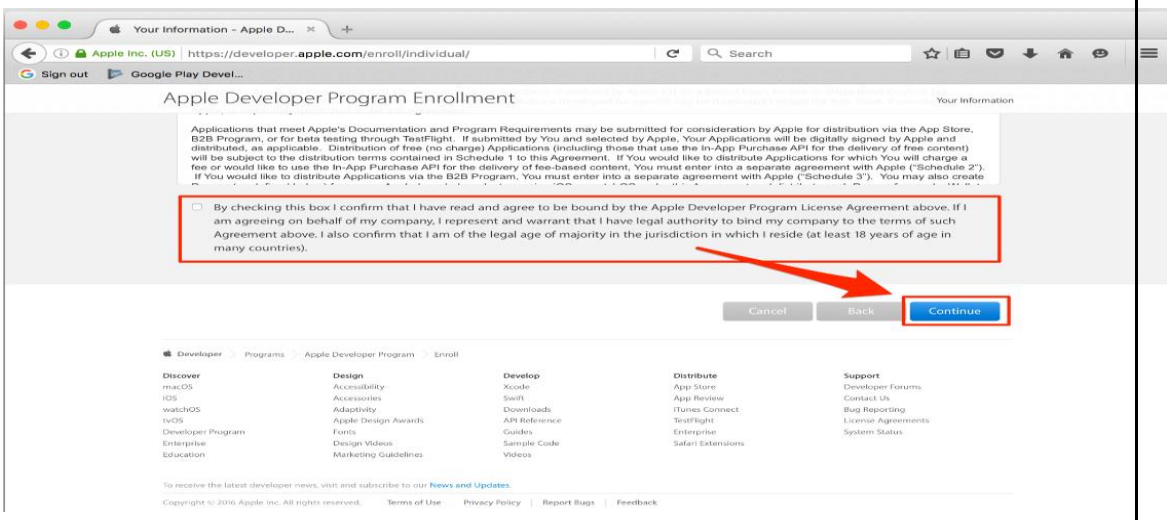
Company/Organization: All legally recognized companies that have a Dun and Bradstreet number can choose this option. It will enable creation and management of multiple user login with different permission capabilities for every login.



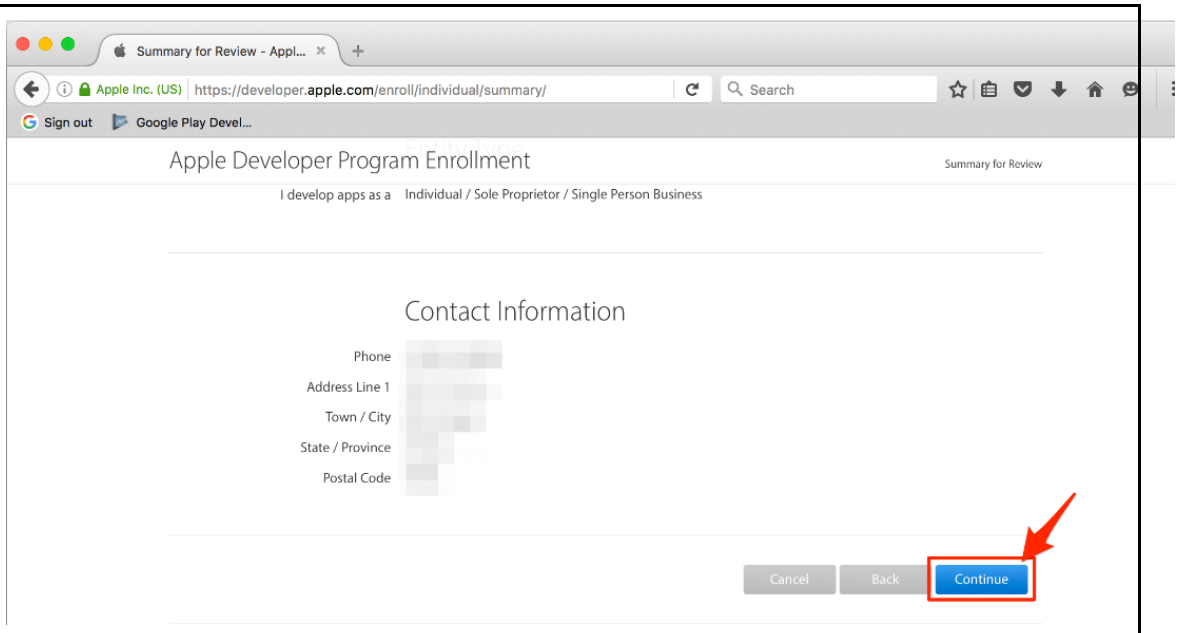
5. Now, fill the Contact Information regarding your Apple Developer Account.



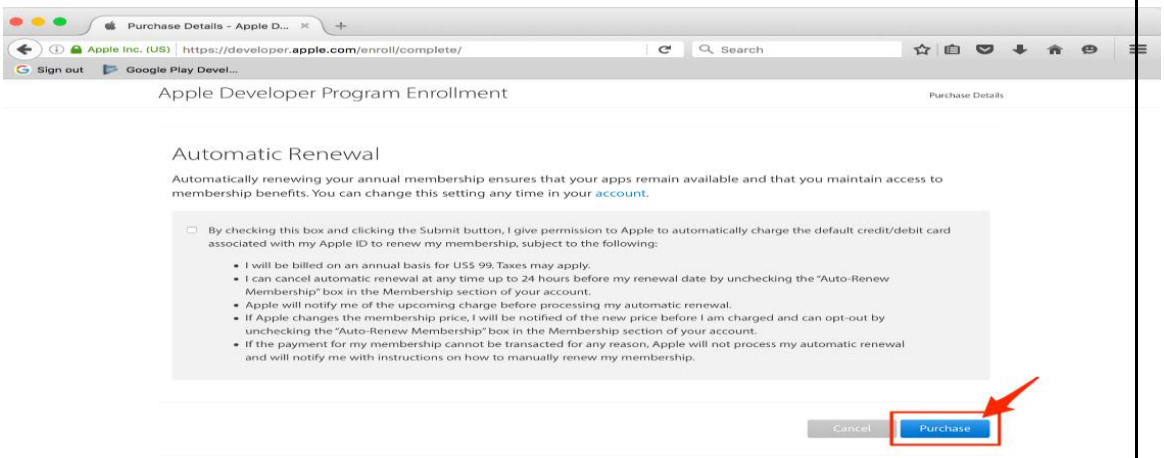
6. At the end of the screen after reading the Apple Developer Program License Agreement simply check the 'By checking this box I confirm that I have read and agree…', box and click Continue.



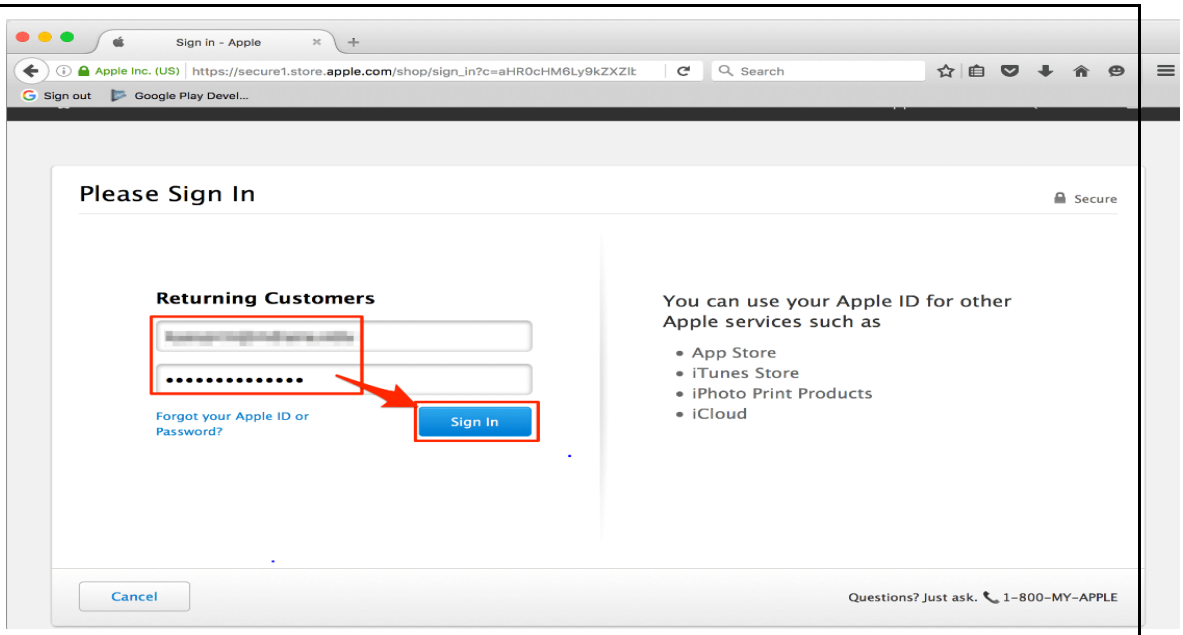
7. Click Continue after confirming the Apple ID Information, Entity Type, and Contact Information.



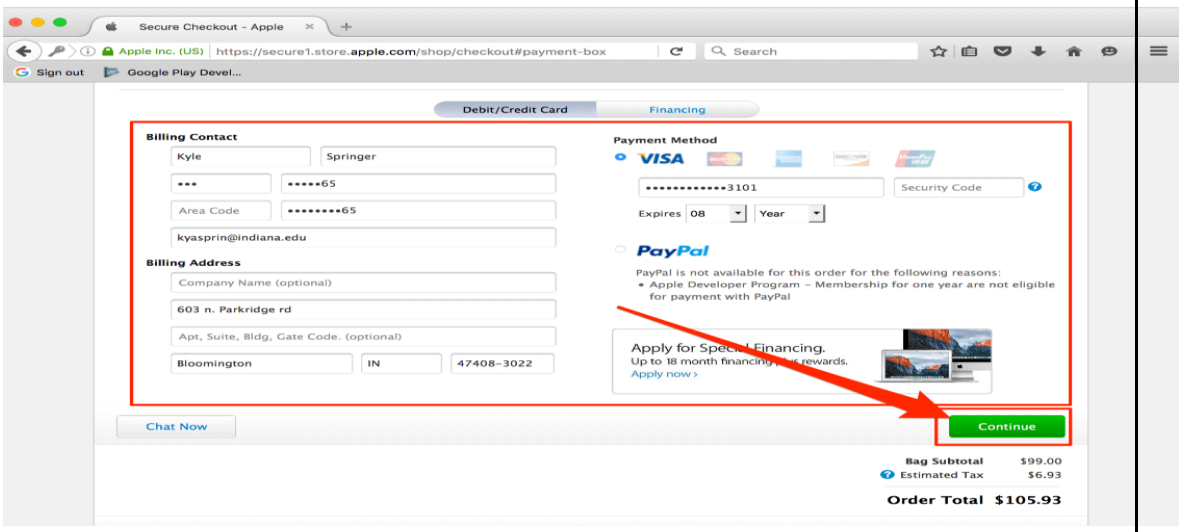
8. Opt the auto-renewal box (this is optional) if you are interested in automatic renewals now click Purchase to enroll and pay for your annual Android DeveloperAccount.



9. You are ready to Sign in with your Apple ID/password.



10. Now enter the required billing information and click Continue for payment completion and setting up your Apple developer Account.



11. At last, fill the remaining information and receive confirmation email within 24 hours from Apple confirming successful creation of Apple Developer Account.

Apple Developer Program Activation Code

Dear Kyle Springer,

To complete your purchase and access your Apple Developer Program benefits, please click on the activation code below.

	Activation Code	Part Number
iOS Developer Program	2649-2PZR-A48Z-Y945	D4521G/A

If you need further assistance, please contact us.

Best regards,

Apple Developer Program Support

Practical Activity 4.2.4: Generating App Release Builds (.ipa, .aab)

Task:

1. You are requested to go to the computer lab to Generate App Release Builds (.ipa, .aab).
2. Apply safety precautions.
3. Referring to the steps provided on task 3, Generate App Release Builds (.ipa, .aab).
4. Present your work to the trainer and whole class.
5. For more clarification, read the key readings 4.2.4. ask questions where necessary.

Key readings 4.2.4: Generating App Release Builds (.ipa, .aab)

Generating App Release Builds (.ipa, .aab) involves creating distributable versions of your app for iOS and Android platforms. Here are the **simple steps**:

✓ **For iOS (.ipa) Build:**

Steps:

1. **Open Project in Xcode**:
   Use Xcode to open your Flutter project's ios/Runner.xcworkspace.
2. **Select App Target**:
   In the top toolbar, choose your app target (e.g., Runner).
3. **Set Scheme to "Release"**:
   Go to **Product > Scheme > Edit Scheme**, and set it to **"Release"** mode.
4. **Archive the App**:
   Go to **Product > Archive** to create an archive of your app.

5. **Export the IPA**:
   - After archiving, select **"Distribute App"**.
   - Choose **"App Store Connect"** or **"Ad Hoc"** based on your distribution needs.
   - Follow the steps to export and save the .ipa file.

✓ For Android (.aab) Build:

Steps:

1. **Open Project in Android Studio**:
   Use **Android Studio** to open your project.
2. **Set Build Variant to "Release"**:
   In the **Build Variants** panel, set the build variant to **"release"**.
3. **Generate Signed Bundle**:
   - Go to **Build > Generate Signed Bundle / APK**.
   - Select **Android App Bundle** and proceed with signing the app to generate the .aab file.

These steps will generate the necessary files to publish your app on the **Apple App Store** (.ipa) and **Google Play Store** (.aab)

Practical Activity 4.2.5: Uploading App bundles (Android & iOS)

Task:

1: You are requested to go to the computer lab to Upload App bundles (Android & iOS).
2: Apply safety precautions
3: Referring to the steps provided on task 3, Upload App bundles (Android & iOS).
4: Present your work to the trainer and whole class
5: For more clarification, read the key readings 4.2.5. ask questions where necessary.

Key readings 4.2.5: Uploading App bundles (Android & iOS)

Steps for Uploading App Bundles to Mobile Application Stores:

✓ For Android (Google Play Store) – Uploading. aab File:

Steps:

1. **Create a Google Play Developer Account**:

   If you don't have one, sign up for a Google Play Developer Account, pay the $25 one-time fee, and set up your account.

2. **Prepare Your App for Release**:

   - Ensure you have your. aab (Android App Bundle) file generated using flutter build appbundle or Android Studio.
   - Update your app's version code and version name in android/app/build.gradle.

3. **Access Google Play Console**:

   Go to the Google Play Console and sign in with your developer account.

4. **Create a New App**:

   - Click **Create App** and fill in the required details like **app name**, **default language**, and **app category**.
   - Accept the **Developer Program Policies** and click **Create**.

5. **Set Up App Details**:

   - Fill in the **Store Listing** details (title, description, screenshots, etc.).
   - Provide a content rating and set the app pricing (free or paid).

6. **Upload the .aab File**:

   - Navigate to the **Release > Production** section.
   - Click **Create New Release**, then **Upload** your **.aab** file.
   - Review any warnings or suggestions from Google Play Console.

7. **Rollout Release**:

   After the file is uploaded, review the release, and when ready, click **Start Rollout to Production** to submit the app for review.

✓ For iOS (Apple App Store) Uploading *.ipa* File:

Steps:

1. **Create an Apple Developer Account**:

   If you don't have one, sign up for an Apple Developer Account and pay the $99 yearly fee.

2. **Prepare Your App for Release**:

   - Generate the **.ipa** file using **Xcode**.
   - Ensure you have set the correct version number and build number in your app.

3. **Access App Store Connect**:

Go to App Store Connect and sign in with your Apple Developer Account.

4. **Create a New App**:
   - Under the **My Apps** section, click **+** to add a new app.
   - Fill in app details like **App Name**, **Primary Language**, **Bundle ID**, and **SKU**.
5. **Upload the .ipa File via Xcode or Transporter**:
   - In Xcode, after archiving the app, select **Distribute App** and choose **App Store Connect** to upload the **.ipa** file directly.
   - Alternatively, you can use **Transporter App** to upload the **.ipa** file manually.
6. **Fill in App Store Details**:
   - Provide app information like the **description**, **keywords**, **support URL**, **screenshots**, and **app icon**.
   - Set up app pricing and availability.
7. **Submit for Review**:

   After filling in the app details and uploading the build, click **Submit for Review**. Apple will review the app, and once approved, you can publish it to the App Store.

Points to Remember

- Preparing Store Assets
  - ✓ App Icon
  - ✓ App Screenshots
- Creating Developer account registration
  - ✓ Google Developer Console
  - ✓ Apple Developer Account
- Generating App Release Builds
  - ✓ .ipa for iOS.
  - ✓ .aab for Android.
- Configuring app setting
  - ✓ App ID (Android & iOS)
- Uploading App bundles (Android & iOS)
- **Creating app Store Assets:**
  - ✓ **Creating App Icon**

    Steps:

    1. Prepare Icon Image
    2. Add flutter_launcher_icons Package
    3. Run the Flutter Launcher Icons Script

- ✓ **Creating App Screenshots**

Steps:

1. Set Up Emulator or Device
2. Capture Screenshot
3. Optional - Use flutter_native_screenshot Plugin

- ✓ **Creating Google Play Developer Account:**

1. **Sign-up**: Go to Google Play Console and click "Create account".
2. **Registration Fee**: Pay a one-time, non-refundable $25 registration fee.
3. **Google Play Developer Console**: Complete the form and return to the Developer Console.
4. **Agreement**: Agree to the Developer Distribution Agreement, then proceed to payment.
5. **Payment**: Enter payment details and pay the $25 fee.
6. **Account Activation**: Wait for an email confirming activation (can take up to a week).
7. **Submit Apps**: Once activated, you can submit apps to the Play Store.

- ✓ **Creating Apple Developer Account:**

1. **Sign-up**: Visit the Apple Developer Enrollment page and click "Start Your Enrollment".
2. **Apple ID**: Sign in with an existing Apple ID or create a new one.
3. **Agreement**: Review and agree to the Apple Developer Agreement, then submit.
4. **Entity Type**: Choose between Individual Developer or Company/Organization.
5. **Contact Information**: Provide accurate contact information.
6. **License Agreement**: Agree to the Apple Developer Program License Agreement, then continue.
7. **Apple ID Info**: Confirm Apple ID, Entity Type, and Contact Information.
8. **Auto-renewal Option**: Opt into auto-renewal (optional).
9. **Annual Fee**: Pay the $99 annual fee to complete enrollment.
10. **Sign in**: Use Apple ID and password to sign in.
11. **Account Activation**: Receive a confirmation email from Apple within 24 hours.

- ✓ Generating App Release Builds (.ipa, .aab) involves different steps:

For iOS (.ipa) Build:

1. Open Project in Xcode
2. Select App Target:
3. Set Scheme to "Release":
4. Archive the App:
5. Export the IPA

For Android (.aab) Build:

1. Open Project in Android Studio:
2. Set Build Variant to "Release":
3. Generate Signed Bundle:

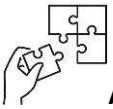
- ✓ **Uploading App bundles (Android & iOS):**

For Android (Google Play Store .aab File)

1. Create a **Google Play Developer Account**.
2. Generate the **.aab** file.
3. Go to the **Google Play Console**.
4. Create a new app in the **Google Play Console**.
5. Complete the **Store Listing** details.
6. Upload the **.aab** file under **Release > Production**.
7. Set pricing and **content rating**.
8. Review and **start rollout to production**.

For iOS (Apple App Store. ipa File)

1. Create an **Apple Developer Account**.
2. Generate the **.ipa** file using **Xcode**.
3. Go to **App Store Connect**.
4. Create a new app in **App Store Connect**.
5. Complete the **App Information** and **Store Listing** details.
6. Upload the **.ipa** file via **Xcode** or **Transporter**.
7. Submit the app for **review**.
8. Wait for **Apple's approval** and release.



Application of learning 4.2.

PQ Company aims to create a fitness tracking mobile app that helps users monitor workouts, set fitness goals, and track their progress over time. The company has hired a developer to design, develop, submit and publish the app on both Google Play an **Apple App Store.**

Indicative content 4.3: Address Post eployment issues

Duration: 5 hrs

Theoretical Activity 4.3.1: Identification of Address post deployment issues

Tasks:

1.You are requested to Answer the following:

i. What tools are being used to monitor and analyze crash reports, and how frequently are they reviewed?
ii. How is the performance of the app currently being measured, and what metrics indicate degradation?
iii. What strategies are in place for optimizing the app store listing to improve visibility and downloads?
iv. What operating system versions and types have been tested for compatibility issues, and what problems have been identified?
v. What is the process for identifying critical issues that require immediate hot fixes, and how are these fixes deployed?

2. Provide your findings and write them on papers.
3. Present the findings to the whole class
4. For more clarification, read the key readings 4.3.1. ask questions where necessary.

Key readings 4.3.1 Identification of Address post deployment issues

Post-deployment issues refer to problems that arise after an app has been released to users. Identifying and addressing these issues quickly is critical to maintaining a smooth user experience and ensuring the app performs optimally.
Below are key strategies used to manage and fix post-deployment issues:

1. Monitor Crash Reports (UXCam, Sentry)

Crash reports provide detailed insights into app crashes, highlighting the exact scenarios and code paths causing these problems. Monitoring crash reports using tools like **UXCam** and **Sentry** enables developers to detect, diagnose, and resolve crashes efficiently.

- **Sentry**: It captures real-time errors and exceptions, offering stack traces, user context, and environment details, which helps developers locate the source of crashes and bugs.
- **UXCam**: This tool combines user experience monitoring with crash analytics, offering insights into how the crash impacts user flow and what user actions led to

it.

2. Performance Degradation

Performance degradation refers to the slowing down or inefficient operation of an app after release. It can manifest as slower load times, lagging UI, or higher battery consumption.

- Tools like **Firebase Performance Monitoring** help track metrics like **app startup time**, **screen render times**, and **network request latency**.
- Identifying specific points where performance declines allows for optimizations, such as optimizing image loading, reducing excessive background tasks, and refining code for better memory management.

3. Applying App Store Optimization (ASO)

App Store Optimization (ASO) is the process of improving the visibility of an app within the app store to drive more downloads. Post-deployment, applying ASO strategies becomes essential to maintaining a competitive edge.

- **Title, Keywords, and Descriptions**: Ensuring the app's name, description, and keywords are optimized for relevant searches helps increase app discoverability.
- **User Ratings and Reviews**: Monitoring and responding to reviews, while also encouraging satisfied users to rate the app, boosts ranking.
- **Screenshots and Icons**: Updating app visuals, including screenshots and app icons, with new features or design updates can engage potential users better.

4. Compatibility Problems

Post-deployment, an app might encounter compatibility issues across various operating systems, devices, or software versions. These issues often arise because new operating system updates introduce changes that affect how the app runs.

- **Based on Operating System Type**: Apps may behave differently on **iOS** and **Android** due to differences in APIs, hardware, and system architecture.
- **Based on Operating System Version**: New **API levels** (for Android) or **iOS versions** may cause incompatibility if the app hasn't been updated to accommodate those changes.

5. Perform Hot Fixing

Hot fixing refers to the process of deploying quick fixes to resolve critical issues without needing a full-scale update. This technique is useful for addressing urgent bugs, crashes, or security vulnerabilities that were not detected during pre-release testing.

- Hot fixes enable rapid correction of issues that severely affect the user experience, minimizing downtime and negative feedback.
- The **hotfix** should be thoroughly tested before release to avoid introducing new bugs.

Practical Activity 4.3.2: Applying of App Store Optimization (ASO)

Task:

1: You are requested to go to the computer lab to Apply of App Store Optimization (ASO).
2: For more clarification, read the key readings 4.3.2. ask questions where necessary.
3: Apply safety precautions.
4: Referring to the steps provided on task 3, Apply of App Store Optimization (ASO).
5: Present your work to the trainer and whole class

Key readings 4.3.2: Applying of App Store Optimization (ASO)

App Store Optimization (ASO) is the process of improving an app's visibility, ranking, and conversion rate in app stores. While ASO principles are similar for **Google Play** and the **Apple App Store**, there are platform-specific differences in how optimization works.

❖ **For Google Play ASO**

Steps:

1. **Keyword Research**: Use tools (AppAnnie, Google Play Console) to identify relevant keywords. Analyze competitors and use long-tail keywords.
2. **Optimizing Title & Description**: Include important keywords in the 50-character title, 80-character short description, and 4,000-character long description. Use bullet points to enhance readability.
3. **App Icon Design**: Create a simple, recognizable icon aligned with your app's branding.
4. **Screenshots & Feature Graphic**: Upload up to 8 high-quality screenshots and a 1024 x 500 px feature graphic. Include a YouTube-linked promo video.
5. **Encouraging Reviews**: Use in-app prompts to request ratings and respond to user reviews.
6. **Optimizing for Installs**: Ensure good app performance, optimize size for faster downloads.
7. **Regular Updates**: Update frequently and clearly communicate changes in the "What's New" section.
8. **Category & Tags**: Choose the right category and add relevant tags in Google Play Console.
9. **Localization**: Translate your app's title, description, and screenshots into multiple languages for local markets.
10. **Tracking & A/B Testing**: Use Google Play Console to track performance and test different elements of your store listing.

❖ **For Apple App Store ASO**

Steps:

1. **Keyword Research**: Use tools (AppTweak, Sensor Tower) to find relevant keywords, focusing on long-tail keywords. Analyze competitor apps.
2. **Optimizing Title & Subtitle**: Use the 30-character title and subtitle to include relevant keywords and describe the app's main benefit.
3. **Optimizing Keyword Field**: Utilize the 100-character keyword field for additional relevant keywords, avoiding repetition.
4. **App Description**: Write a clear, engaging description with calls to action. Though not indexed for search, it influences conversions.
5. **App Icon Design**: Ensure the icon is simple, memorable, and aligned with your app's branding.
6. **Screenshots & App Preview Video**: Upload up to 10 screenshots and an optional 15–30 second preview video that highlights key features.
7. **Encouraging Reviews**: Use in-app prompts to request reviews at appropriate moments, and respond to feedback to improve perception.
8. **Regular Updates**: Publish frequent updates and communicate changes clearly in the "What's New" section to engage users.
9. **Category & Subcategory**: Choose relevant categories and subcategories to increase app visibility.
10. **Localization**: Localize your app's title, description, keywords, and screenshots for different regions.
11. **Tracking & A/B Testing**: Use App Store Connect to monitor performance and run A/B tests for icons, screenshots, and descriptions.

Practical Activity 4.3.3: Performing Hot fixing

Task:

1: You are requested to go to the computer lab to Perform Hot fixing in mobile application.
2: For more clarification, read the key readings 4.3.3. ask questions where necessary
3: Apply safety precautions
4: Referring to the steps provided on task 3, to Perform Hot fixing in mobile application.
5: Present your work to the trainer and whole class

Key readings 4.3.3: Performing Hot fixing

Hotfixing in a mobile application refers to the process of quickly applying a small, critical update to fix an issue in the application, often without going through the full release cycle. It is especially useful for resolving urgent bugs that affect functionality or user experience.

Here are the typical steps involved in performing a hotfix for a mobile application:

1. Identify the Issue

- **Monitor and analyze logs**: Use tools like Crashlytics, Firebase, or Sentry to identify bugs and crashes in the app.
- **Reproduce the bug**: Ensure you can consistently replicate the issue to understand the cause and scope of the fix.

2. Branching for Hotfix

- **Create a hotfix branch**: In your version control system (Git), create a new branch from the main or stable branch, specifically for the hotfix (e.g., hotfix/bug-fix-123).

bash

Copy code

git checkout -b hotfix/bug-fix-123 main

3. Apply the Fix

- **Fix the bug**: Apply the necessary code changes to resolve the issue.
- **Test locally**: Ensure the fix works as intended by testing it on various devices and screen sizes.

4.Build and Test

- **Run unit tests**: If you have automated unit tests, ensure that they all pass after applying the hotfix.
- **Perform manual testing**: Test the application manually for any regressions, focusing on the area where the bug was reported.
- **Build APK/IPA**: Build the updated mobile app for Android (APK) and iOS (IPA) platforms.
  - For Android, use Android Studio or Gradle to generate an APK.
  - For iOS, use Xcode to generate an IPA file.

5. Deploy the Hotfix

- **Submit to app stores**: Depending on the severity of the hotfix, submit the updated version to the Google Play Store and Apple App Store.
  - **Google Play Store**: You can opt for a **staged rollout** to release the update to a percentage of users first, then expand the release after ensuring no issues arise.
  - **Apple App Store**: You can choose an **expedited review** if the hotfix is

critical.

- **Monitor for issues**: After deployment, continue monitoring analytics and logs to ensure that the hotfix resolves the issue and does not introduce new bugs.

6. Merge Hotfix into Main Branch

- Once the hotfix has been successfully deployed and verified, merge the hotfix branch back into the main or master branch to ensure the fix is included in future releases.

bash
Copy code
git checkout main
git merge hotfix/bug-fix-123
git push origin main

7. Document the Hotfix

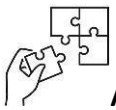
- Keep a record of the changes, including a description of the bug, the solution, and the deployment steps, for future reference.

Points to Remember

- ✓ Tools are being used to monitor and analyze crash reports are: **Sentry, Crashlytics (Firebase),** Bugsnag, **New Relic,** Rollbar, AppDynamics
- ✓ Strategies are in place for optimizing the app store listing to improve visibility and downloads are: **Keyword Optimization,** Engaging Visual Assets, App Description, App Description, Localized Content, App Ratings & Reviews
- ✓ Operating system versions and types have been tested for compatibility issues Are: Identify Target Platforms, Emulators and Real Devices, Focus Areas for Compatibility Testing, Automated and Manual Testing, Regression Testing
- ✓ Process for identifying critical issues that require immediate hot fixes, and how are these fixes deployed.

While performing the Address PostDeployment Issues, pass on the following steps:

- ✓ Monitor Crash Reports
- ✓ Perform Degradation
- ✓ Apply the app Store Optimization (ASO)
- ✓ Perform the Compatibility Problems
- ✓ Perform Hot Fixing



Application of learning 4.3.

RedCompany has launched a **calculator app** that performs basic arithmetic operations.

While the initial launch was successful, users reported several issues that impact the app's usability and performance. The task is to maintain user satisfaction and prevent negative reviews, prioritize hotfixes**,** optimize performance, and ensure the app works smoothly across a wide range of devices.

Learning outcome 4 end assessment

Written assessment

I. **Circle the letter corresponding to the correct answer:**

1. Which file format is used for iOS app packages?
    a. APK
    b. IPA
    c. AAB
    d. JAR
2. What does Hot Restart do in mobile development?
    a. Restarts the app without losing the current state
    b. Compiles code at runtime
    c. Displays real-time changes without a full restart
    d. Generates a new app bundle
3. In the app submission process, which of the following is NOT a required asset?
    a. App Icon
    b. App Screenshots
    c. User Manual
    d. App Promotional Materials
4. Which Google tool is used for app distribution?
    a. Google Drive
    b. Google Play Console
    c. Google Forms
    d. Google Analytics
5. What is the purpose of the App ID in mobile app development?
    a. To identify the app on the developer's computer
    b. To provide unique identification for the app-on-app stores
    c. To track app downloads
    d. To optimize app performance
6. What does ASO stand for in the context of mobile applications?
    a. App Store Optimization
    b. Application Security Operations
    c. Application System Organization
    d. App Software Output
7. Which service is commonly used to monitor crash reports after an app is deployed?
    a. Firebase
    b. UXCam
    c. Sentry
    d. All of the above

II.Fill the empty space by using the correct word listed in the bracket.

1. The two main types of builds in mobile app development are _____ and _____.

(Beta Build, Alpha Build, Debug Build, Release Build, Prototype Build)

2. The _____ file format is used to package Android applications for distribution on the Google Play Store. (IPA, AAB, EXE, APK, JAR)

3. In iOS development, the final app package is typically distributed as an _____ file. (IPA, AAB, EXE, APK, JAR)

4. The process of_____ involves optimizing the app's listing to improve visibility and attract more downloads. (App Development, App Promotion, App Store Optimization (ASO), User Testing, Marketing Strategy)

5. A developer must register with _____ to publish apps on the Apple App Store. (Google Play Console, Apple Developer Program, Microsoft Developer Network, Android Developer Portal, Amazon Developer Console)

6. To monitor application performance and user experience, developers can use tools like _____and _____. (Firebase and Sentry, Excel and Google Docs, Photoshop and Illustrator, Slack and Zoom, Trello and Asana)

7. After deployment, issues related to _____ can arise if the app is not compatible with certain devices or operating system versions. (Security Vulnerabilities, compatibility Problems, Usability Issues, Performance Degradation, User Interface Design)

IV. Match the following terms to their description and write letters in an empty place of answers.

Answers	Term	Description
1……...	1) Beta Builds	A) Contains polished, stable versions of apps for public use
2…….	2) Release Builds	B) Allows developers to gather feedback from real users before an official release
3……	3) iOS App Submission Process	C) Requires a one-time registration fee and tax/banking information for app sales payouts
4……..	4) Google Developer Account	D) Translates code into machine language at runtime, just before execution
5…….	5) Just-in-Time (JIT) Compilation	E) Uses App Store Connect and requires IPA file submission
6……..	6) AOT Compilation	F) Preferred for production environments due to faster execution and lower memory usage
7……	7) JIT Compilation	G) Helps identify specific compatibility issues across different devices and OS versions
		H) Preferred in development environments for dynamic features and runtime optimizations

Practical assessment

At **TechSphere Solutions**, mobile application development using the **Flutter SDK** follows a collaborative, streamlined process where trainees play an active role. Mobile developers generate and compile cross-platform installable files for both iOS and Android, while the DevOps team automates build pipelines and manages app store submissions across all platforms. Product managers coordinate between teams to ensure client requirements are met, while the QA team ensures bug-free releases. The marketing and ASO team prepares store assets and optimizes app visibility. Post-deployment, trainees in the support team monitor crash reports and user feedback, addressing issues with hotfixes or rollbacks when necessary. Continuous App Store Optimization (ASO) and crash monitoring ensure optimal app performance and discoverability, delivering quality Flutter apps efficiently on Windows or macOS, or Linux systems.

References

https://www.geeksforgeeks.org/what-is-ide/
https://www.w3schools.com/
https://stackoverflow.com/
https://chatgpt.com/g/g-RGr8YRENd-chart-gpt-3
Miola, A. (2020). Flutter Complete Reference: Create Beautiful, Fast and Native Apps for Any Device. KDP: Amazon LLC.
Napoli, M. L. (2019). Beginning Flutter: A Hands On Guide to App Development 1st Edition.
Rose, R. (2022). Flutter and Dart Cookbook 1st Edition. Calfornia: O'Reilly Media.

RTB | RWANDA TVET BOARD

October, 2024