DEPARTMENT OF COMPUTER SCIENCE & ENGINERING (AI&ML)

UI DESIGN-FLUTTER LAB

Lab Manual

Subject Code : AM507PC

Regulation : R22/JNTUH

Academic Year : 2024-2025

III B. TECH I SEMESTER

MALLA REDDY COLLEGE OF ENGINEERING

(Formerly CM Engineering College)

Approved by AICTE & Permanently Affiliated to JNTUH: ISO 9001:2015 Certified Institution

DEPARTMENT OF COMPUTER SCIENCE & ENGINERING (AI&ML)

INSTITUTE VISION & MISSION

VISION

To become an institute of excellence by creating high quality and innovating engineering andmanagement professionals who would take the world into their stride through sustainable growth in technolog and management.

MISSION

To instill moral values and promote technological, intellectual and ethical environment to the students with an in-depth and exceptional education that makes them employment ready asper the emerging trends in industry and to invoke the desire of innovation as a process of life-long learning for a successful career in engineering and management.

DEPARTMENT VISION & MISSION

VISION

To evolve the department of computer science & engineering as a centre of academicexcellence with latest technologies to transform students into innovative global leaders.

MISSION

M1: To produce competitive graduates having creative skills and ethical values to succeed intheir fields as well as the foundation for life-long learning.

M2: By promoting research and development activities in collaboration with reputed industriesand laboratories.

M3: To analyze design, and develop high-quality software systems using the appropriatetheory, principles, tools and processes.

DEPARTMENT OF COMPUTER SCIENCE & ENGINERING (AI&ML)

PROGRAM EDUCATIONAL OBJECTIVES (PEOS):

A graduate of the Computer Science and Engineering Program should:

PEO1	**Program Educational Objective1: (PEO1)** The Graduates will provide solutions to difficult and challenging issues in their profession by applying computer science and engineering theory and principles.
PEO2	**Program Educational Objective2 :(PEO2)** **The** Graduates have successful careers in computer science and engineering fields or will be able to successfully pursue advanced degrees.
PEO3	**Program Educational Objective3: (PEO3)** The Graduates will communicate effectively, work collaboratively and exhibit high levels of Professionalism, moral and ethical responsibility.
PEO4	**Program Educational Objective4 :(PEO4)** **The** Graduates will develop the ability to understand and analyse Engineering issues in a broader perspective with ethical responsibility towards sustainable development.

PROGRAM OUTCOMES (POS):

PO1	**Engineeringknowledge**:Applytheknowledgeofmathematics,science,engineering Fundamentals andanengineeringspecializationtothesolutionofcomplexengineeringproblems.
PO2	**Problem analysis**: Identify, formulate, review research literature, and analyze complex engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences, and engineering sciences.
PO3	**Design/development of solutions**: Design solutions for complex engineering problems and design system components or processes that meet the specified needs with appropriate consideration for the public health and safety, and the cultural, societal, and environmental considerations.
PO4	**Conduct investigations of complex problems**: Use research-based knowledge and research methods including design of experiments, analysis and interpretation of data, and synthesis of the information to provide valid conclusions.

PO5	**Modern tool usage**: Create, select, and apply appropriate techniques, resources, and modern engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.
PO6	**The engineer and society**: Apply reasoning informed by the contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to the professional engineering practice.
PO7	**Environment and sustainability**: Understand the impact of the professional engineering Solutions in societal and environmental contexts, and demonstrate the knowledge of, and need for sustainable development.
PO8	**Ethics**: Apply ethical principles and commit to professional ethics and responsibilities and norms of the engineering practice.
PO9	**Individual and team work**: Function effectively as an individual, and as a member or leader In diverse teams, and in multi-disciplinary settings.
PO10	**Communication**: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as, being able to comprehend and write effective reports and design documentation, make effective presentations, and give and receive clear instructions.
PO11	**Project management and finance**: Demonstrate knowledge and understanding of the Engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.
PO12	**Life-long learning**: Recognize the need for, and have the preparation and ability to engage in independent and life-long learning in the broadest context of technological change.

PROGRAM SPECIFIC OUTCOMES(PSOS):

PSO1	**Problem Solving Skills** – Graduate will be able to apply computational techniques and software principles to solve complex engineering problems pertaining to software engineering.
PSO2	**Professional Skills** – Graduate will be able to think critically, communicate effectively, and collaborate in teams through participation in co and extra-curricular activities.
PSO3	**Successful Career** – Graduates will possess a solid foundation in computer science and engineering that will enable them to grow in their profession and pursue lifelong learning through post-graduation and professional development.

DEPARTMENT OF COMPUTER SCIENCE & ENGINERING (AI&ML)

Course Objectives:

- Learns to Implement Flutter Widgets and Layouts
- Understands Responsive UI Design and with Navigation in Flutter
- Knowledge on Widges and customize widgets for specific UI elements, Themes
- Understand to include animation apart from fetching data

Course Outcomes:

- Implements Flutter Widgets and Layouts
- Responsive UI Design and with Navigation in Flutter
- Create custom widgets for specific UI elements and also Apply styling using themes and custom styles.
- Design a form with various input fields, along with validation and error handling
- Fetches data and write code for unit Test for UI components and also animation

DEPARTMENT OF INFORMATION TECHNOLOGY

Course Name: **UI DESIGN-FLUTTER LAB** **Course Code**: AM**507PC**

Year/Semester: **III/I** **Regulation: R22**

S. No	List of Experiments	Page No.
1	a) Install Flutter and Dart SDK. b) Write a simple Dart program to understand the language basics.	
2	a) Explore various Flutter widgets (Text, Image, Container, etc.). b) Implement different layout structures using Row, Column, and Stack widgets.	
3	a) Design a responsive UI that adapts to different screen sizes. b) Implement media queries and breakpoints for responsiveness.	
4	a) Set up navigation between different screens using Navigator. b) Implement navigation with named routes.	
5	a) Learn about stateful and stateless widgets. b) Implement state management using set State and Provider.	
6	a) Create custom widgets for specific UI elements. b) Apply styling using themes and custom styles.	
7	a) Design a form with various input fields. b) Implement form validation and error handling.	
8	a) Add animations to UI elements using Flutter's animation framework. b) Experiment with different types of animations (fade, slide, etc.).	
9	a) Fetch data from a REST API. b) Display the fetched data in a meaningful way in the UI.	
10	a) Write unit tests for UI components. b) Use Flutter's debugging tools to identify and fix issues.	

FACULTY **HOD**

1. a) Install Flutter and Dart SDK.

Pre-installation Requirements

- Windows 10 operating system installed (Flutter will work on Windows 7 SP1 and later versions).

At least 1.65 GB of free disk space (Additional free storage is needed for other tools and IDEs, if not already installed).

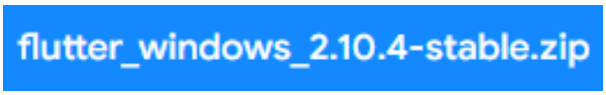
- Windows Powershell 5.0 or newer.
- Git for Windows version 2.0 or newer (Optional).
- Android Studio installed.
- Visual Studio 2022 with C++ (Optional).

How to Install and Configure Flutter SDK on Windows 10

After meeting all requirements, you can begin installing and configuring Flutter SDK. In today's tutorial, you will be installing a fixed installation of Flutter SDK, without using Git.

Step 1: Download Flutter SDK

Download the Flutter SDK package by clicking on the following button on the webpage.

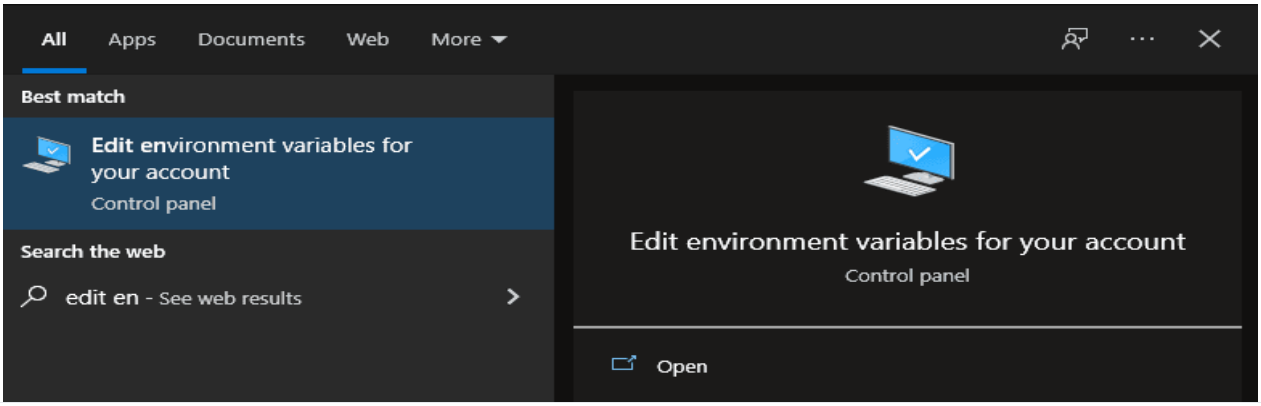


Step 2: Extract the Files

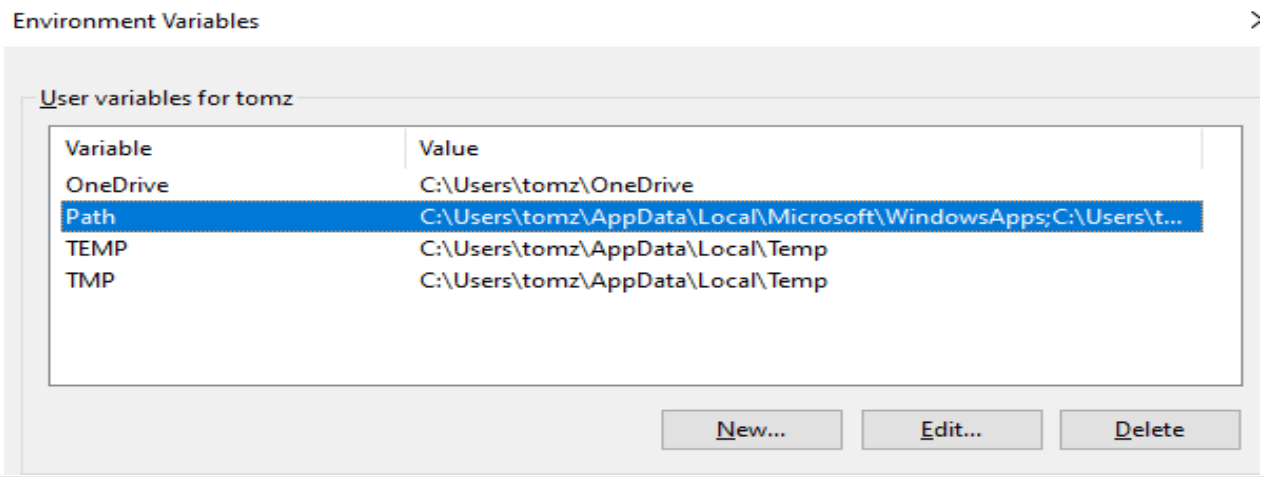
Extract the downloaded zip file and move it to the desired location you want to install Flutter SDK. Do not install it in a folder or directory that requires elevated privileges, (such as *C:\Program Files*) to ensure the program runs properly. For this tutorial, it will be stored in *C:\development\flutter*.

Step 3: Update Path Variable for Windows PowerShell

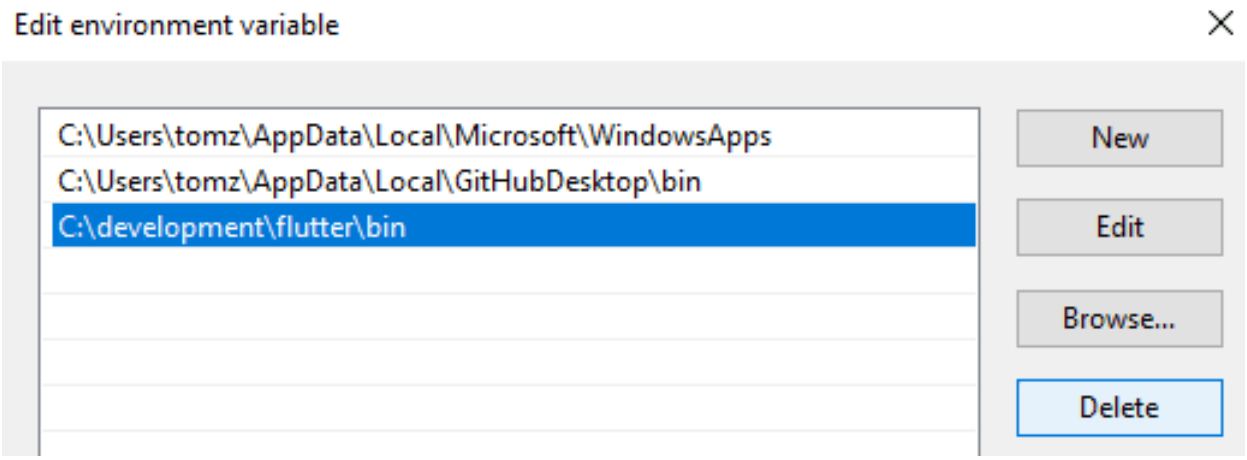
Next, you need to update your Path environment variable to run Flutter commands in Windows consoles PowerShell and Command Prompt (CMD). First, click the Start button and type to search for and then click on **Edit environment variables for your account**.



Under User variables, click on and highlight **Path**. Click **Edit**.



On the next screen, click **New** and add the full path to your *flutter\bin* directory. For this guide, it is shown below. Click OK on both windows to enable running Flutter commands in Windows consoles.



Step 4: Confirm Installed Tools for Running Flutter

In CMD, run the *flutter doctor* command to confirm the installed tools along with brief descriptions.

```
C:\Users\tomz>flutter doctor
Running "flutter pub get" in flutter_tools... 8,9s
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, 2.10.4, on Microsoft Windows [Version 10.0.19041.746], locale en-US)
[X] Android toolchain - develop for Android devices
X Unable to locate Android SDK.
Install Android Studio from: https://developer.android.com/studio/index.html
On first launch it will assist you in installing the Android SDK components.
(or visit https://flutter.dev/docs/get-started/install/windows#android-setup for detailed instructions).
If the Android SDK has been installed to a custom location, please use
```

'flutter config --android-sdk' to update to that location.

[√] Chrome - develop for the web

[X]] Visual Studio - develop for Windows

X Visual Studio not installed; this is necessary for Windows development.

Download at https://visualstudio.microsoft.com/downloads/.

Please install the "Desktop development with C++" workload, including all of its default components

[!] Android Studio (not installed)

[√] Connected device (2 available)

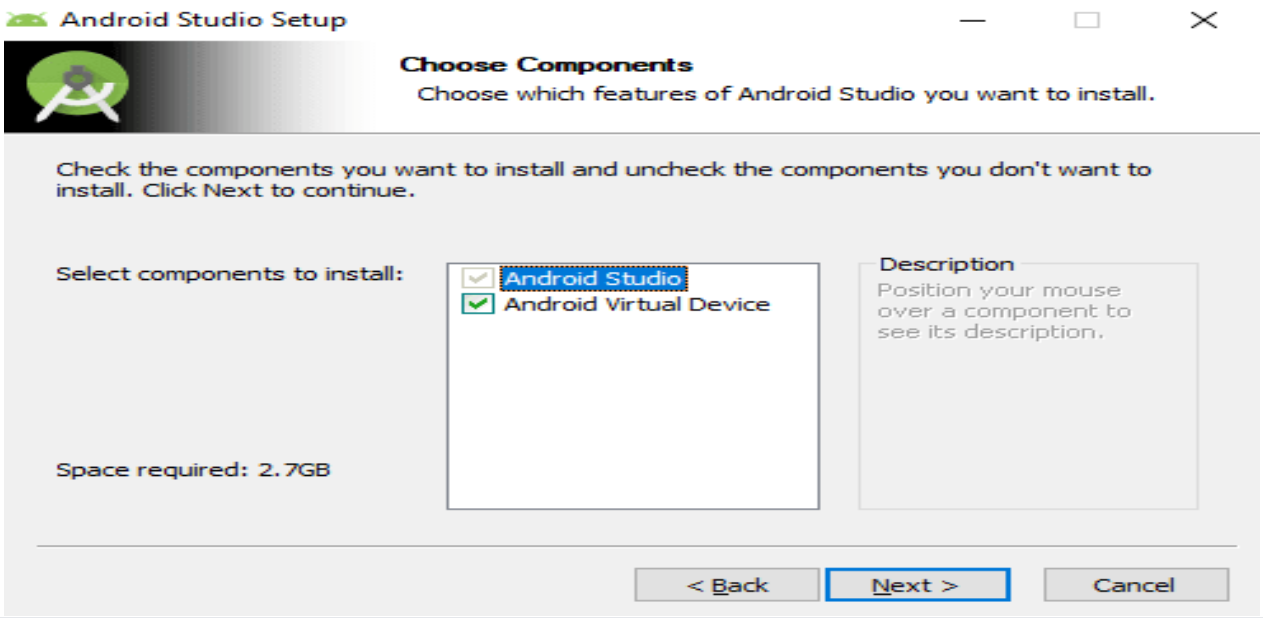
[√] HTTP Host Availability

! Doctor found issues in 3 categories.

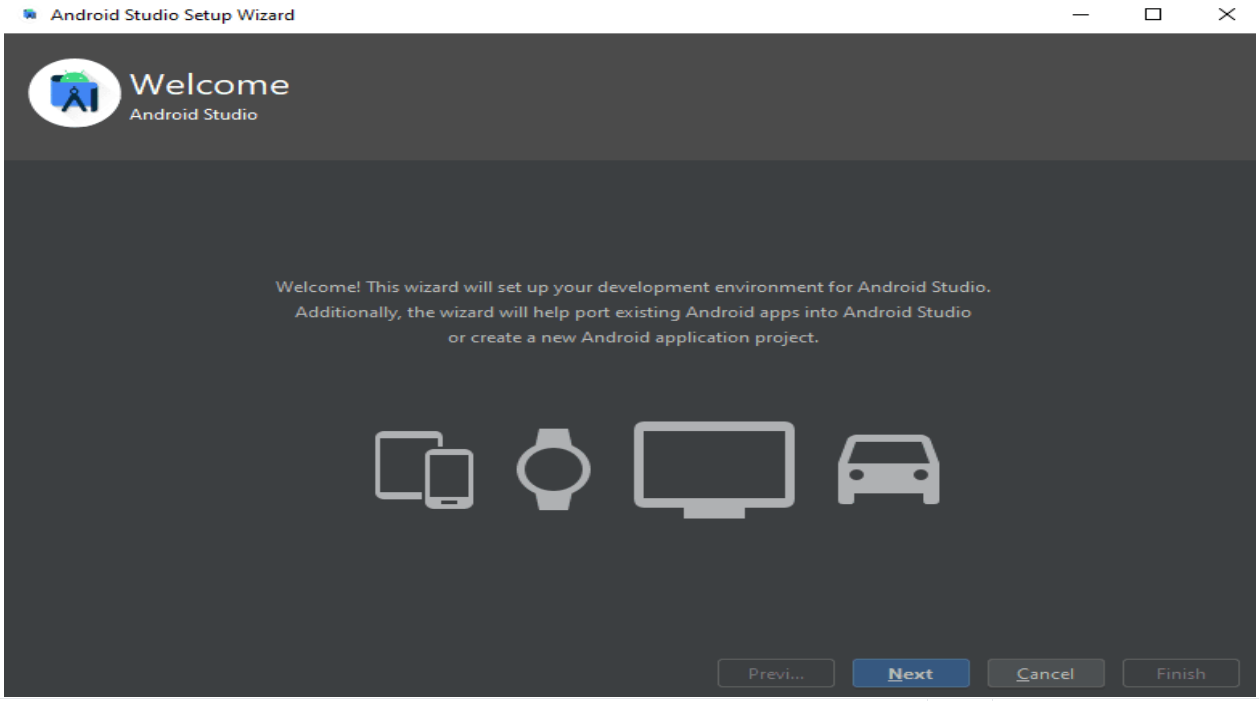
As visible, several components still need to be installed to complete the installation.

Step 5: Download and Install Android Studio

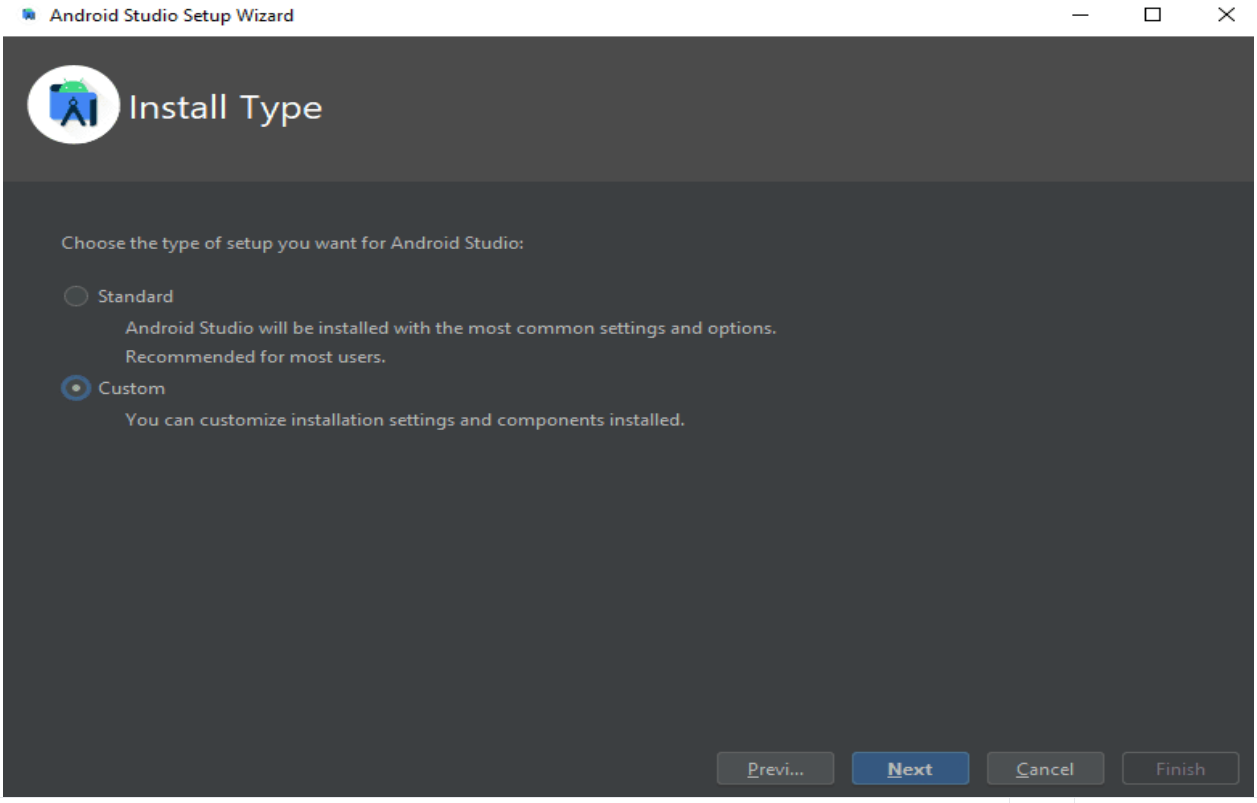
Continue by downloading Android Studio. In the setup, unless you have specific requirements, you can click Next on all screens leaving the default settings. Ensure that the Android Virtual Device option is selected on the Choose Components screen so that you can have an Android emulator running for Android app development.



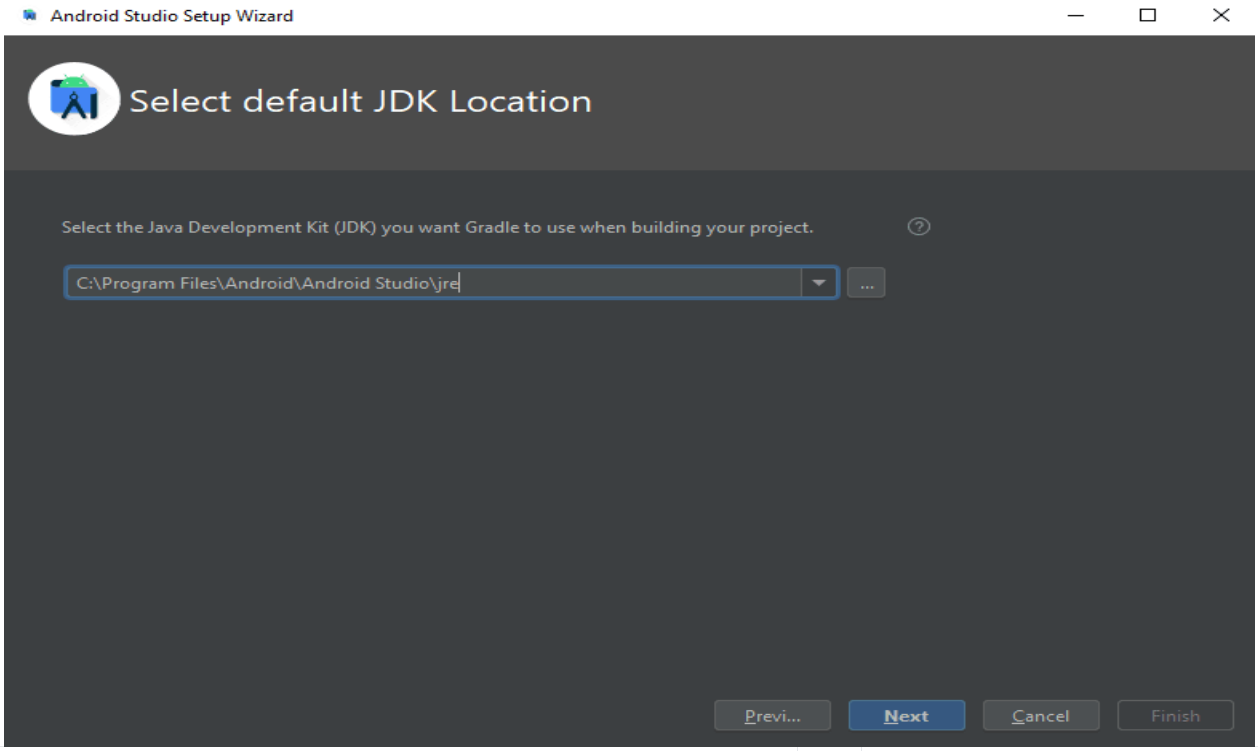
Afterward, Android Studio Setup Wizard will start and you can proceed by clicking **Next**.



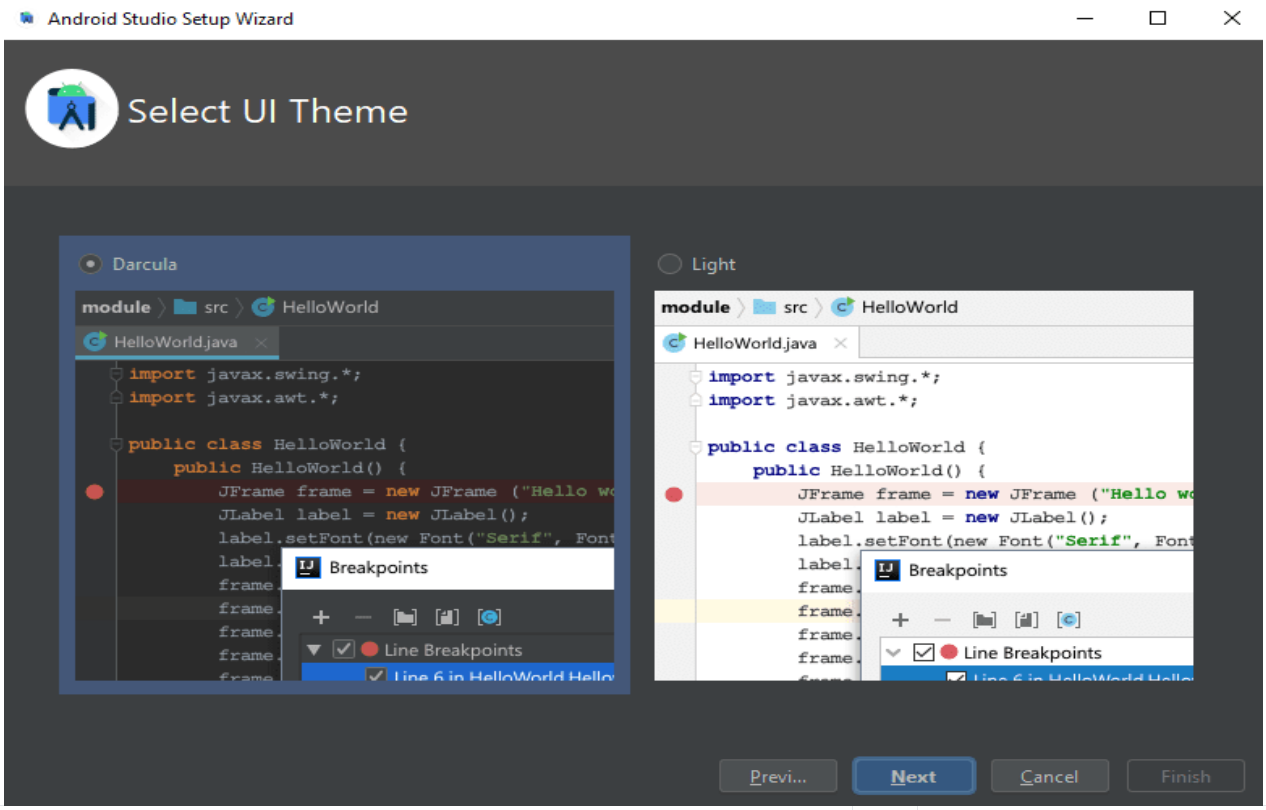
On the Install Type screen, select Custom and click **Next**.



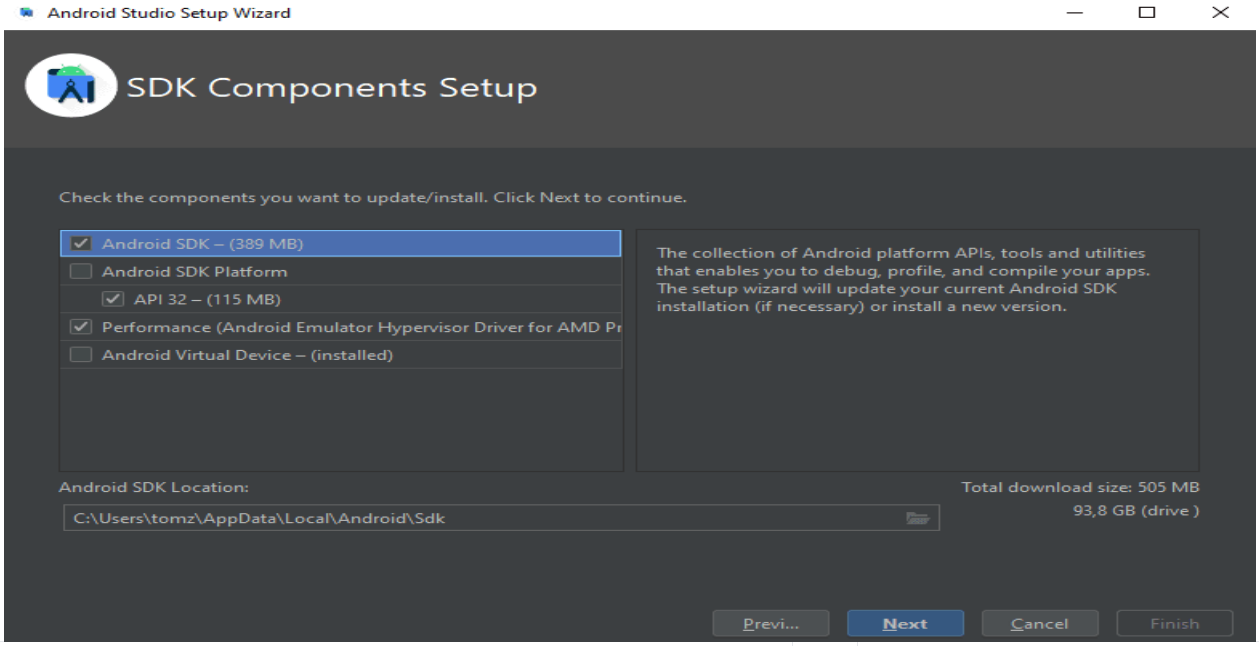
Select the installation location or leave the default path and click **Next**.



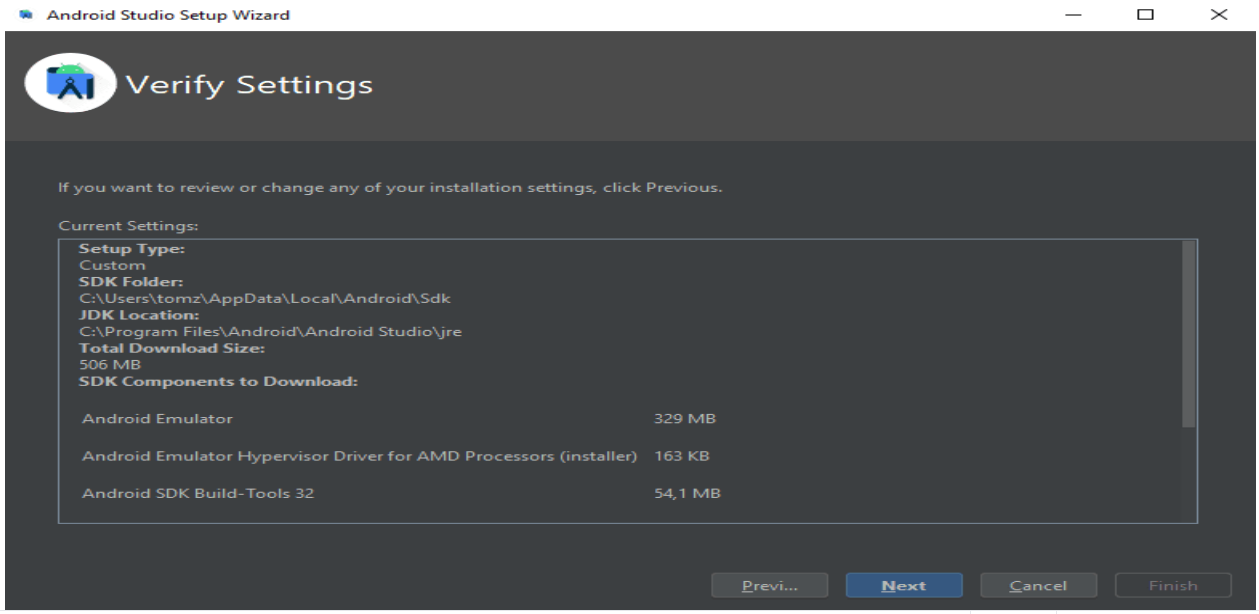
Select your UI theme and click **Next**.



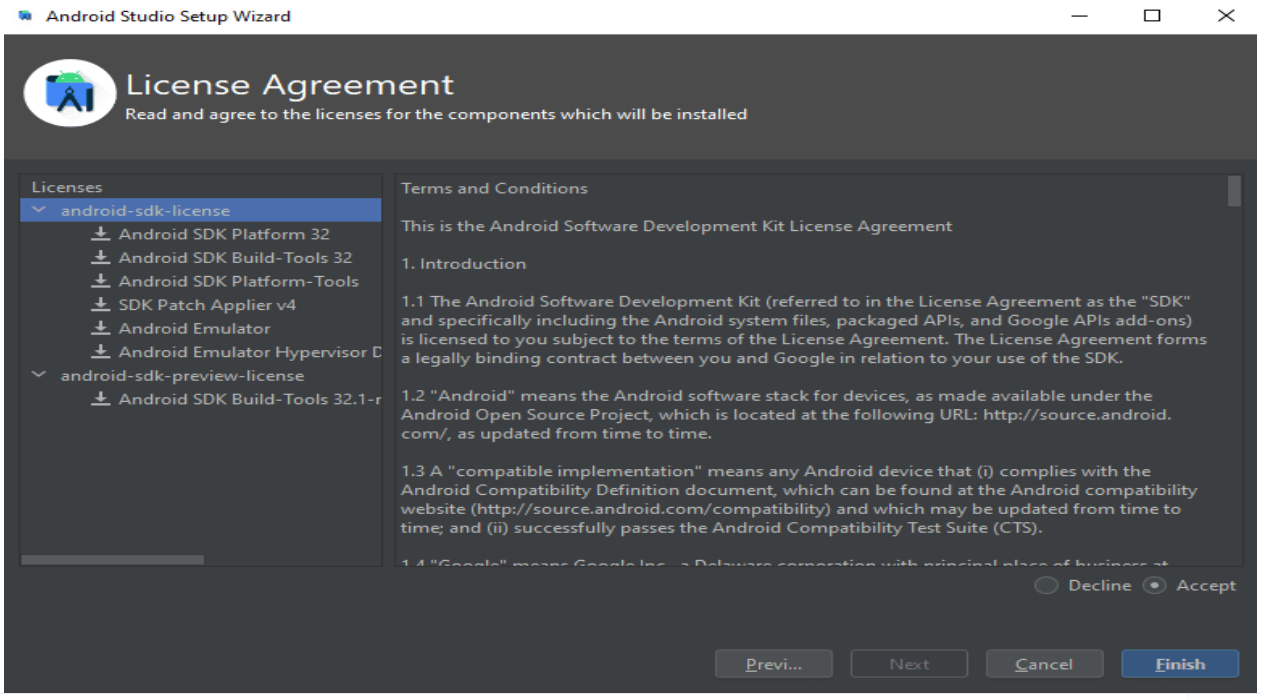
Select your SDK components and click **Next**.



Verify the selections and click **Next**.

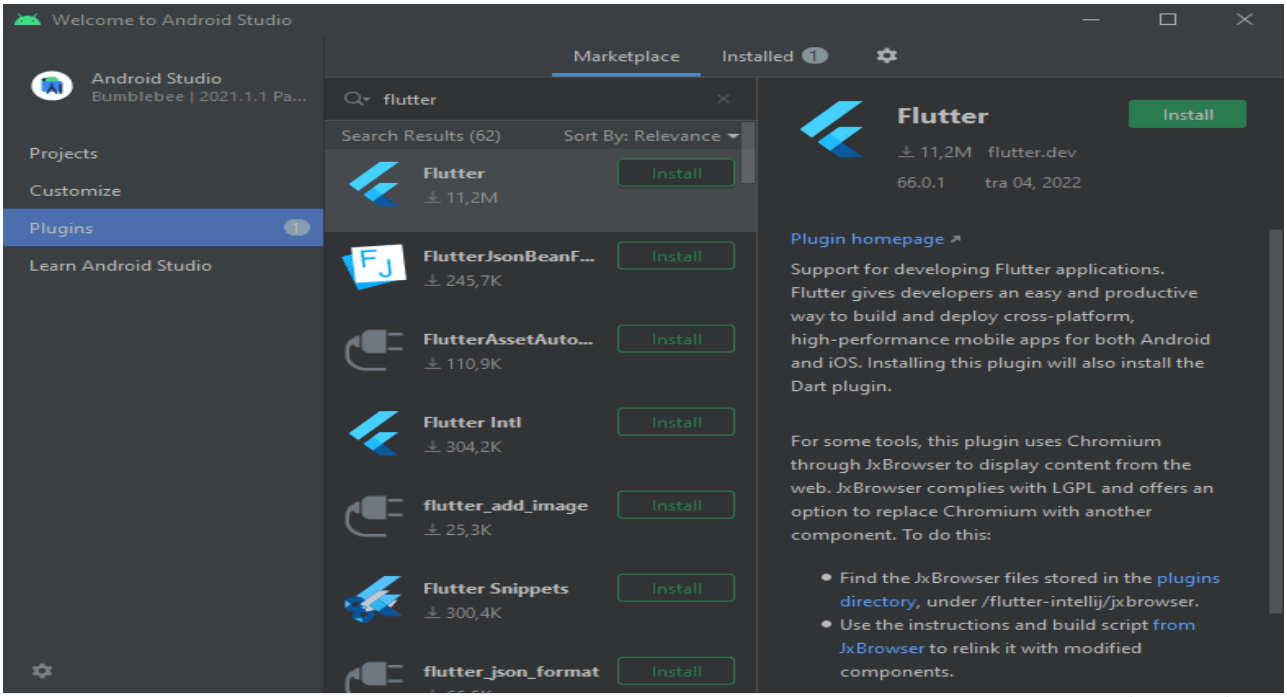


On the next screen, accept the License Agreement and click **Finish**.

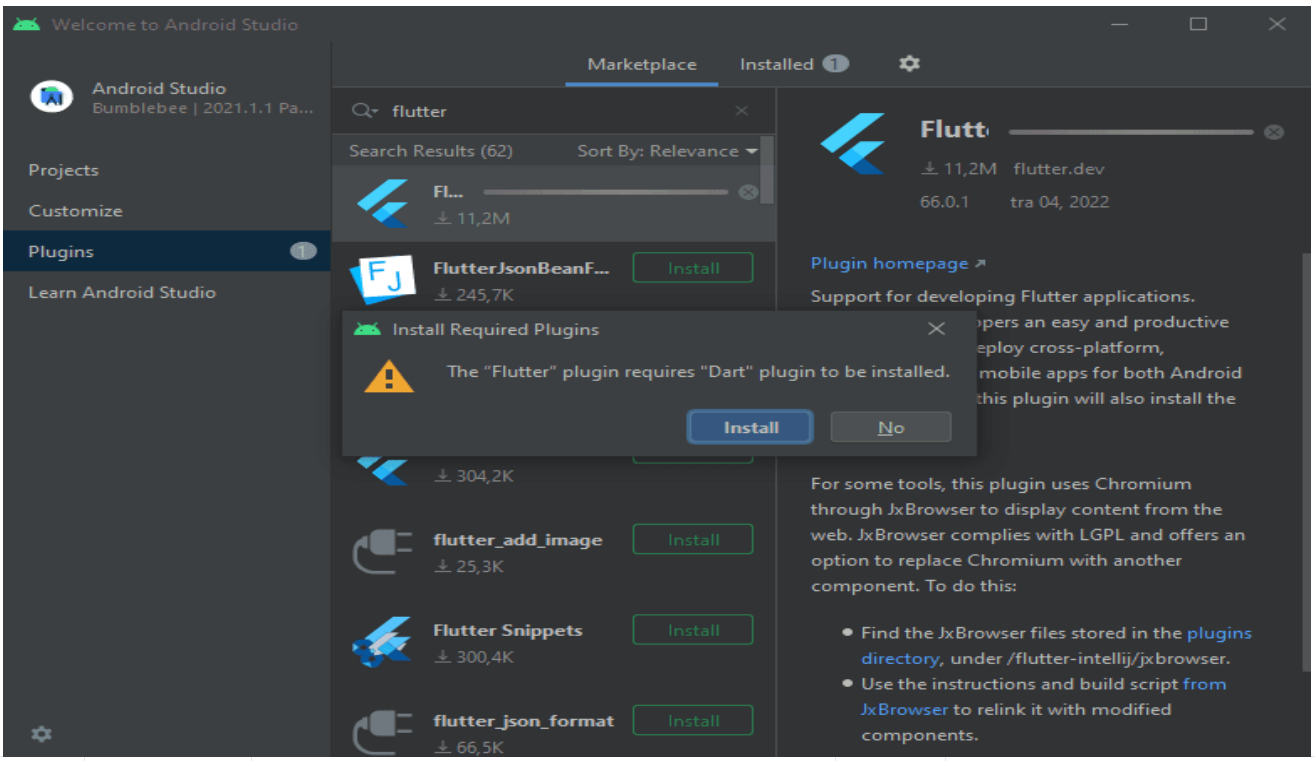


The download of the components will start and Android Studio install. Once completed, click **Finish**.

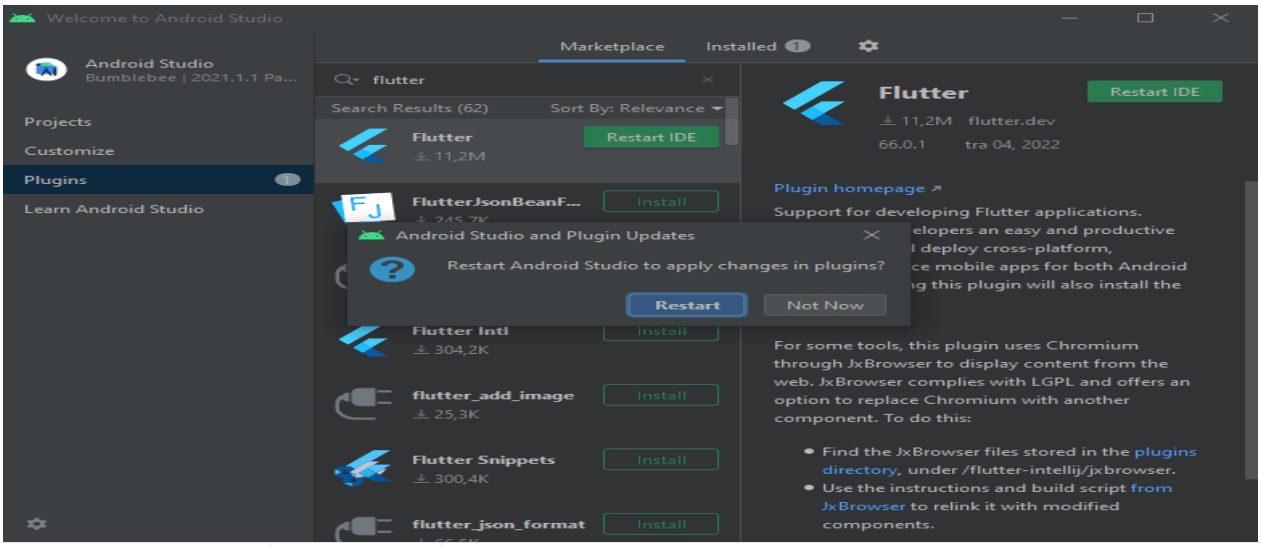
After the installation, start Android Studio. On the left side, click **Plugins**. Search for Flutter and click **Install** to install the Flutter plugin.



It will also prompt you to install Dart, a programming language used to create Flutter apps. Click **Install** at the prompt.



Finally, click **Restart IDE** so that the plugin changes are applied. Click **Restart** at the prompt to confirm this action.



Afterward, run the *flutter doctor*command in CMD to confirm the Android Studio installation.

C:\Users\tomz>flutter doctor

Doctor summary (to see all details, run flutter doctor -v):

[√] Flutter (Channel stable, 2.10.4, on Microsoft Windows [version 10.0.19041.746), locale en-US)

[!] Android toolchain - develop for Android devices (Android SDK version 32.1.0-rc1)

! Some Android licenses not accepted. To resolve this, run: flutter doctor --android-licenses

[√] Chrome - develop for the web

[X] Visual Studio - develop for Windows

X Visual Studio not installed; this is necessary for Windows development.

Download at https://visualstudio.microsoft.com/downloads/.

Please install the "Desktop development with C++- workload, including all of its default components

[√] Android Studio (version 2021.1)

[√] Connected device (2 available)

[√] HTTP Host Availability

! Doctor found issues in 2 categories.

Android Studio was successfully installed, however, it finds an issue with Android licenses. This issue is fairly common and is mitigated by running the following command in CMD.

flutter doctor --android-licenses

When asked, input **y** to all prompts, to accept licenses.

C:\Users\tomz>flutter doctor --android-licenses

5 of 7 SDK package licenses not accepted. 100% Computing updates...

Review licenses that have not been accepted (y/N)? y

Running the *flutter doctor* command again shows the issue resolved.

C:\Users\tomz>flutter doctor

Doctor summary (to see all details, run flutter doctor -v):

[√] Flutter (Channel stable, 2.10.4, on Microsoft Windows [Version 10.0.19041.746], locale en-US)

[√] Android toolchain - develop for Android devices (Android SDK version 32.1.0-rc1)

[√] Chrome - develop for the web

[X] Visual Studio - develop for Windows

X Visual Studio not installed; this is necessary for Windows development.

Download at https://visualstudio.microsoft.com/downloads/.

Please install the "Desktop development with C++" workload, including all of its default components

[√] Android Studio (version 2021.1)

[√] Connected device (2 available)

[√] HTTP Host Availability

! Doctor found issues in 1 category.

b) Write a simple Dart program to understand the language basics.
Aim: To perform simple Dart program to understand the language basics

Basic Dart Program

This is a simple dart program that prints **Hello World** on screen. Most programmers write the Hello World program as their first program.

```
void main() {
print("Hello World!");
}
```

Basic Dart Program Explained

- void main() is the starting point where the execution of your program begins.
- Every program starts with a main function.
- The curly braces {} represent the beginning and the ending of a block of code.
- print("Hello World!"); prints Hello World! on screen.
- Each code statement must end with a semicolon.

Basic Dart Program For Printing Name

```
void main()
{
var name ="John";
print(name);
}
```

Basic Dart Program To Join One Or More Variables

Here **$variableName** is used to join variables. This joining process in dart is called string interpolation.

```
void main(){
var firstName ="John";
var lastName ="Doe";
print("Full name is $firstName $lastName");
}
```

Dart Program For Basic Calculation

Performing addition, subtraction, multiplication, and division in dart.

```
void main() {
int num1 =10; //declaring number1
int num2 =3; //declaring number2

// Calculation
int sum = num1 + num2;
int diff = num1 - num2;
```

```
int mul = num1 * num2;
double div = num1 / num2; // It is double because it outputs number with decimal.

// displaying the output
print("The sum is $sum");
print("The diff is $diff");
print("The mul is $mul");
print("The div is $div");
}
```

Create Full Dart Project

It's nice to work on a single file, but if your project gets bigger, you need to manage configurations, packages, and assets files. So creating a dart project will help you to manage this all.

```
dart create <project_name>
```

This will create a simple dart project with some ready-made code.

Steps To Create Dart Project

- Open folder location on command prompt/terminal.
- Type `dart create project_name` (For E.g. dart create first_app)
- Type `cd first_app`
- Type `code .` to open project with visual studio code
- To check the main dart file go to **bin/first_app.dart** and edit your code.

Run Dart Project

First, open the project location on the command/terminal and run the project with this command.

```
dart run
```

Result: Hence we performed simple Dart program to understand the language basics

2.a) Explore various Flutter widgets (Text, Image, Container, etc.).
Aim: To perform various Flutter widgets (Text, Image, Container, etc.).
Program:
Flutter provides a wide range of widgets to build rich and interactive user interfaces for mobile, web, and desktop applications. Here's a brief exploration of some commonly used widgets:
Text: The Text widget is used to display text on the screen. It allows you to customize the text's style, such as font size, color, alignment, etc.

```
Text(
 'Hello, Flutter!',
 style: TextStyle(
  fontSize: 20,
  color: Colors.blue,
  fontWeight: FontWeight.bold,
 ),
```

network, or memory.

```
Image.asset('assets/images/flutter_logo.png'),
```

nents with various styling

```
Container(
  width: 100,
  height: 100,
  color: Colors.red,
  child: Center(
    child: Text(
      'Container',
      style: TextStyle(
        color: Colors.white,
      ),
```

izontally aligned layouts.

```
Row(
  children: [
    Icon(Icons.star),
    Text('5.0'),
  ],
),
```

ating vertically aligned layouts.

```
  Column(
    children: [
      Text('First item'),
      Text('Second item'),
```

have a large number of items to

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

des astandard title, leading, and

```
AppBar(
  title: Text('My App'),
  actions: [
    IconButton(
      icon: Icon(Icons.search),
      onPressed: () {
        // Add search functionality
      },
    ),
  ],
),
```

ystem allows for easy
l UIs efficiently.
, etc.).
k widgets.
lgets in Flutter

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

class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
return MaterialApp(
home: Scaffold(
appBar: AppBar(
title: Text('Layout Examples'),

```
),
body: Column(
mainAxisAlignment: MainAxisAlignment.center,
crossAxisAlignment: CrossAxisAlignment.center,
children: [
Text('Column Example'),
Row(
mainAxisAlignment: MainAxisAlignment.spaceEvenly,
children: [
Container(
color: Colors.blue,
height: 50,
width: 50,
),
Container(
color: Colors.green,
height: 50,
width: 50,
),
Container(
color: Colors.red,
height: 50,
width: 50,
),
],
),
Stack(
alignment: Alignment.center,
children: [
Container(
color: Colors.yellow,
height: 100,
width: 100,
),
Text(
'Stack Example',
style: TextStyle(color: Colors.white),
),
],
),
],
),
),
);
}
}
```

- This Flutter code creates a simple app with different layout structures:
- **Row:** Three containers aligned horizontally with equal space between them.
- **Column:** Three containers aligned vertically with space between them.
- **Stack:** A container stacked on top of another container with positioning.

Result: Thus we Implementing different layout structures using Row, Column, and Stack widgets in Flutter

3. a) Design a responsive UI that adapts to different screen sizes.

Aim: To design a responsive UI that adapts to different screen sizes in flutter

To design a responsive UI in Flutter that adapts to different screen sizes, you can use Flutter's built-in layout widgets such as Row, Column, Container, Expanded, Flexible, and MediaQuery. Here's an example of how you can create a responsive UI in Flutter:

Program:

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

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

class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
return MaterialApp(
title: 'Responsive UI',
theme: ThemeData(
primarySwatch: Colors.blue,
),
home: MyHomePage(),
);
}
}

class MyHomePage extends StatelessWidget {
@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(
title: Text('Responsive UI Example'),
),
body: LayoutBuilder(
builder: (context, constraints) {
if (constraints.maxWidth > 600) {
// Wide layout
return WideLayout();
} else {
// Narrow layout
return NarrowLayout();
}
},
),
);
}
}

class WideLayout extends StatelessWidget {
@override
Widget build(BuildContext context) {
return Row(
mainAxisAlignment: MainAxisAlignment.center,
```

```
children: [
Container(
width: 200,
height: 200,
color: Colors.red,
child: Center(
child: Text('Left Content'),
),
),
Container(
width: 400,
height: 200,
color: Colors.green,
child: Center(
child: Text('Right Content'),
),
),
],
);
}
}

class NarrowLayout extends StatelessWidget {
@override
Widget build(BuildContext context) {
return Column(
mainAxisAlignment: MainAxisAlignment.center,
children: [
Container(
width: double.infinity,
height: 200,
color: Colors.red,
child: Center(
child: Text('Top Content'),
),
),
SizedBox(height: 20),
Container(
width: double.infinity,
height: 200,
color: Colors.green,
child: Center(
child: Text('Bottom Content'),
),
),
],
);
}
}
```

- The **LayoutBuilder** widget is used to dynamically choose between different layouts based on the available width.
- When the screen width is greater than 600, a wide layout is displayed with content side by side. Otherwise, a narrow layout is displayed with content stacked vertically.

- Depending on the layout, different widgets such as Row, Column, and Container are used to arrange and display content.
- You can adjust the width and height of the containers, as well as other properties, to fit your specific design requirements.

Result: Thus we designed a responsive UI that adapts to different screen sizes in flutter

b) Implement media queries and breakpoints for responsiveness.

Aim: To Implement media queries and breakpoints for responsiveness in Flutter

In Flutter, you can implement responsiveness using MediaQuery and breakpoints. MediaQuery provides information about the current app screen such as size and orientation, which you can use to make layout decisions. Breakpoints are specific screen sizes at which you may want to adjust your layout.

Program:

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

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

class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
return MaterialApp(
home: Scaffold(
appBar: AppBar(
title: Text('Responsive App'),
),
body: LayoutBuilder(
builder: (context, constraints) {
// Define your breakpoints
double screenWidth = MediaQuery.of(context).size.width;
if (screenWidth > 600) {
return WideLayout();
} else {
return NarrowLayout();
}
},
),
),
);
}
}

class WideLayout extends StatelessWidget {
@override
Widget build(BuildContext context) {
return Center(
child: Container(
width: 300,
```

```
height: 200,
color: Colors.blue,
child: Text(
'Wide Layout',
style: TextStyle(fontSize: 20, color: Colors.white),
),
),
);
}
}

class NarrowLayout extends StatelessWidget {
@override
Widget build(BuildContext context) {
return Center(
child: Container(
width: 200,
height: 300,
color: Colors.red,
child: Text(
'Narrow Layout',
style: TextStyle(fontSize: 20, color: Colors.white),
),
),
);
}
}
```

MediaQuery.of(context).size.width is used to get the width of the screen.
LayoutBuilder is used to rebuild the layout when the constraints change (e.g., screen orientation or size).
WideLayout and **NarrowLayout** represent different layouts for wide and narrow screens, respectively.
Based on the screen width, the appropriate layout is chosen.
You can adjust the breakpoints and layout accordingly to fit your design requirements.

Result: Thus we implemented media queries and breakpoints for responsiveness in Flutter

4. a) Set up navigation between different screens using Navigator.

Aim: To navigation between different screens using Navigator in flutter.

To set up navigation between different screens using Navigator in Flutter, you typically follow these steps:

1. Define your screens as separate StatelessWidget or StatefulWidget classes.
2. Use MaterialApp or CupertinoApp as the root widget to provide navigation context.
3. Use Navigator to manage navigation between screens, typically by pushing and popping routes onto/from a stack.

Program:

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

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

```
class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
return MaterialApp(
title: 'Flutter Navigation Example',
initialRoute: '/',
routes: {
'/': (context) => HomeScreen(),
'/second': (context) => SecondScreen(),
},
);
}
}

class HomeScreen extends StatelessWidget {
@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(
title: Text('Home Screen'),
),
body: Center(
child: ElevatedButton(
onPressed: () {
Navigator.pushNamed(context, '/second');
},
child: Text('Go to Second Screen'),
),
),
);
}
}

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

Result: Hence we perform navigation between different screens using Navigator in flutter.

b) Implement navigation with named routes.

Aim: To Implement navigation with named routes in flutter

In Flutter, you can implement navigation with named routes by using the MaterialApp widget and defining a map of named routes

Program:

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

class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
return MaterialApp(
// Define the initial route
initialRoute: '/',
// Define named routes
routes: {
'/': (context) => HomeScreen(),
'/second': (context) => SecondScreen(),
'/third': (context) => ThirdScreen(),
},
);
}
}

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

```
}
}

class SecondScreen extends StatelessWidget {
@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(title: Text('Second Screen')),
body: Center(
child: ElevatedButton(
onPressed: () {
// Navigate to the third screen using named route
Navigator.pushNamed(context, '/third');
},
child: Text('Go to Third Screen'),
),
),
);
}
}

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

- MaterialApp is used as the root widget.
- The routes property is used to define a map of named routes, where the keys are route names and the values are builder functions that return the widgets for those routes.
- Navigator.pushNamed(context, routeName) is used to navigate to a named route.
- Navigator.pop(context) is used to go back to the previous screen.

Result: Thus we Implemented navigation with named routes in flutter

5. A.Learn about stateful and stateless widgets

1. **Stateful Widgets**:
   - Stateful widgets are dynamic widgets that can change their appearance or behavior over time.
   - They maintain state, meaning they can hold data and update it when needed, causing the UI to rebuild accordingly.

- Stateful widgets are typically used when the UI elements need to change based on user interactions, data changes, or other external factors.
- They consist of two classes: a StatefulWidget class and a State class.
- The StatefulWidget class is immutable and creates an instance of the State class, which manages the widget's state and controls how it's displayed.
- When the state of a stateful widget changes, the build method of the associated State object is called, and the UI is rebuilt to reflect the new state.

Program:

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

class CounterWidget extends StatefulWidget {
@override
_CounterWidgetState createState() => _CounterWidgetState();
}

class _CounterWidgetState extends State<CounterWidget> {
int _counter = 0;

void _incrementCounter() {
setState(() {
_counter++;
});
}

@override
Widget build(BuildContext context) {
return Container(
padding: EdgeInsets.all(16.0),
child: Column(
mainAxisAlignment: MainAxisAlignment.center,
children: <Widget>[
Text(
'Counter Value:',
style: TextStyle(fontSize: 20.0),
),
Text(
'$_counter',
style: TextStyle(fontSize: 36.0),
),
SizedBox(height: 20.0),
ElevatedButton(
onPressed: _incrementCounter,
child: Text('Increment'),
),
],
),
);
}
}
```

2. **Stateless Widgets**:

- Stateless widgets, on the other hand, are static widgets that cannot change their state once they are built.
- They do not have any internal state and are purely based on the information provided at the time of creation.
- Stateless widgets are used for UI elements that do not need to change, such as static text, icons, buttons, etc.
- Since they do not have any state, they are more efficient and lightweight compared to stateful widgets.
- Stateless widgets are implemented using a single StatelessWidget class, and they typically have a build method that returns the UI components.
- Once a stateless widget is built, it remains the same until it's rebuilt explicitly by calling the setState method or when its parent widget is rebuilt.

Program:

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

class StaticTextWidget extends StatelessWidget {
@override
Widget build(BuildContext context) {
return Container(
padding: EdgeInsets.all(16.0),
child: Text(
'Hello, World!',
style: TextStyle(fontSize: 24.0),
),
);
}
}
```

6. a) Create custom widgets for specific UI elements.

Aim: To create custom widgets for specific UI elements in flutter

- To create custom widgets for specific UI elements in Flutter, you can follow these steps:
- Identify the UI Element: Determine the specific UI element for which you want to create a custom widget. For example, let's say you want to create a custom button.
- Create a Stateless or Stateful Widget: Depending on the complexity and interactivity of the UI element, decide whether to create a StatelessWidget or a StatefulWidget.
- Implement the Widget: Implement the custom widget by extending StatelessWidget or StatefulWidget. Define its appearance and behavior as required.

- Customize Properties: Expose parameters (also known as props or properties) that allow users of the custom widget to customize its appearance or behavior.
- Use the Custom Widget: Finally, utilize the custom widget in your app wherever necessary.

Program

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

class CustomButton extends StatelessWidget {
final String text;
final VoidCallback onPressed;
final Color color;
final Color textColor;

const CustomButton({
Key? key,
required this.text,
required this.onPressed,
this.color = Colors.blue,
this.textColor = Colors.white,
}) : super(key: key);

@override
Widget build(BuildContext context) {
return ElevatedButton(
onPressed: onPressed,
style: ElevatedButton.styleFrom(
primary: color,
),
child: Text(
text,
style: TextStyle(
color: textColor,
),
),
);
}
}
```

b) Apply styling using themes and custom styles.

Aim: To apply styling using themes and custom styles in flutter

In Flutter, you can apply styling using themes and custom styles by defining themes in your MaterialApp and applying custom styles to your widgets

Program:

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

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

class MyApp extends StatelessWidget {
```

```
@override
Widget build(BuildContext context) {
return MaterialApp(
title: 'Themed App',
theme: ThemeData(
// Define your primary colors
primaryColor: Colors.blue,
accentColor: Colors.orange,
// Define text themes
textTheme: TextTheme(
headline1: TextStyle(fontSize: 24.0, fontWeight: FontWeight.bold),
bodyText1: TextStyle(fontSize: 16.0),
),
),
home: HomePage(),
);
}
}

class HomePage extends StatelessWidget {
@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(
title: Text('Home'),
),
body: Center(
child: Column(
mainAxisAlignment: MainAxisAlignment.center,
children: [
Text(
'Welcome to My App',
style: Theme.of(context).textTheme.headline1,
),
SizedBox(height: 20.0),
ElevatedButton(
onPressed: () {},
child: Text('Click Me'),
),
],
),
),
);
}
}
```

Result: Thus we applied styling using themes and custom styles in flutter

b) Apply styling using themes and custom styles.

Aim: To apply styling using themes and custom styles in flutter**.**

In Flutter, you can apply styling to your widgets using themes and custom styles. Themes allow you to define a set of consistent styles that can be applied throughout your app. Custom styles enable you to define specific styles for individual widgets or groups of widgets.

Program:

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

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

class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
return MaterialApp(
theme: ThemeData(
primaryColor: Colors.blue,
accentColor: Colors.orange,
fontFamily: 'Roboto',
// Add more theme properties as needed
),
home: MyHomePage(),
);
}
}

class MyHomePage extends StatelessWidget {
@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(
title: Text('My App'),
),
body: Center(
child: Text(
'Hello, World!',
style: TextStyle(
color: Theme.of(context).accentColor,
fontSize: 24.0,
fontWeight: FontWeight.bold,
// Add more text styles as needed
),
),
),
);
}
}
```

7. a) Design a form with various input fields.

Program

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

void main() {
```

```
runApp(MyApp());
}

class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
return MaterialApp(
title: 'Form Example',
home: Scaffold(
appBar: AppBar(
title: Text('Form Example'),
),
body: MyForm(),
),
);
}
}

class MyForm extends StatefulWidget {
@override
_MyFormState createState() => _MyFormState();
}

class _MyFormState extends State<MyForm> {
final _formKey = GlobalKey<FormState>();

// Define variables to store form field values
String _name = '';
String _email = '';
String _password = '';

@override
Widget build(BuildContext context) {
return Padding(
padding: EdgeInsets.all(16.0),
child: Form(
key: _formKey,
child: Column(
crossAxisAlignment: CrossAxisAlignment.start,
children: <Widget>[
TextFormField(
decoration: InputDecoration(labelText: 'Name'),
validator: (value) {
if (value == null || value.isEmpty) {
return 'Please enter your name';
}
return null;
},
onChanged: (value) {
setState(() {
_name = value;
});
},
```

```
),
TextFormField(
decoration: InputDecoration(labelText: 'Email'),
validator: (value) {
if (value == null || value.isEmpty) {
return 'Please enter your email';
}
return null;
},
onChanged: (value) {
setState(() {
_email = value;
});
},
),
TextFormField(
obscureText: true,
decoration: InputDecoration(labelText: 'Password'),
validator: (value) {
if (value == null || value.isEmpty) {
return 'Please enter a password';
}
return null;
},
onChanged: (value) {
setState(() {
_password = value;
});
},
),
Padding(
padding: const EdgeInsets.symmetric(vertical: 16.0),
child: ElevatedButton(
onPressed: () {
if (_formKey.currentState!.validate()) {
ScaffoldMessenger.of(context).showSnackBar(
SnackBar(
content: Text('Form is valid!'),
),
);
// Perform operations with form data here
// For example: send data to server, etc.
}
},
child: Text('Submit'),
),
),
],
),
),
);
}
}
```

b) Implement form validation and error handling.

In Flutter, you can implement form validation and error handling using the Form widget along with TextFormField widgets for input fields.

import 'package:flutter/material.dart';

Program:

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

class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
return MaterialApp(
title: 'Form Validation Demo',
home: Scaffold(
appBar: AppBar(
title: Text('Form Validation Demo'),
),
body: MyForm(),
),
);
}
}

class MyForm extends StatefulWidget {
@override
_MyFormState createState() => _MyFormState();
}

class _MyFormState extends State<MyForm> {
final _formKey = GlobalKey<FormState>();
TextEditingController _nameController = TextEditingController();
TextEditingController _emailController = TextEditingController();

@override
Widget build(BuildContext context) {
return Form(
key: _formKey,
child: Padding(
padding: EdgeInsets.all(16.0),
child: Column(
crossAxisAlignment: CrossAxisAlignment.start,
children: <Widget>[
TextFormField(
controller: _nameController,
validator: (value) {
if (value.isEmpty) {
return 'Please enter your name';
}
return null;
```

```
},
decoration: InputDecoration(
labelText: 'Name',
),
),
TextFormField(
controller: _emailController,
validator: (value) {
if (value.isEmpty) {
return 'Please enter your email';
}
// You can add more complex validation for email here
return null;
},
decoration: InputDecoration(
labelText: 'Email',
),
),
Padding(
padding: const EdgeInsets.symmetric(vertical: 16.0),
child: ElevatedButton(
onPressed: () {
if (_formKey.currentState.validate()) {
// If the form is valid, proceed with submission
ScaffoldMessenger.of(context).showSnackBar(
SnackBar(content: Text('Form submitted')));
// Add your submission logic here
}
},
child: Text('Submit'),
),
),
],
),
),
);
}

@override
void dispose() {
// Clean up the controller when the widget is removed from the widget tree
_nameController.dispose();
_emailController.dispose();
super.dispose();
}
}
```

8. a) Add animations to UI elements using Flutter's animation framework.

Program

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

void main() {
```

```
runApp(MyApp());
}

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

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

class _MyHomePageState extends State<MyHomePage>
with SingleTickerProviderStateMixin {
AnimationController _animationController;
Animation<double> _animation;

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

// Define animation controller
_animationController = AnimationController(
vsync: this,
duration: Duration(seconds: 1), // Duration of the animation
);

// Define animation
_animation = Tween<double>(
begin: 0.0,
end: 1.0,
).animate(_animationController);

// Start the animation
_animationController.forward();
}

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

@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(
```

```
title: Text('Flutter Animation Example'),
),
body: Center(
child: AnimatedBuilder(
animation: _animationController,
builder: (context, child) {
return Opacity(
opacity: _animation.value, // Opacity changes from 0.0 to 1.0
child: Container(
width: 200,
height: 200,
color: Colors.blue,
child: Center(
child: Text(
'Animated Text',
style: TextStyle(
fontSize: 20.0,
color: Colors.white,
),
),
),
),
);
},
),
),
);
}
}
```

b) Experiment with different types of animations (fade, slide, etc.).

In Flutter, you can create different types of animations using various techniques and widgets. Here's a basic example demonstrating how to implement fade and slide animations using the **AnimatedOpacity** and **SlideTransition** widgets respectively.

Program

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

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

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

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

```
}

class _MyHomePageState extends State<MyHomePage> {
bool _isVisible = true;

void toggleVisibility() {
setState(() {
_isVisible = !_isVisible;
});
}

@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(
title: Text('Animation Demo'),
),
body: Center(
child: Column(
mainAxisAlignment: MainAxisAlignment.center,
children: [
AnimatedOpacity(
duration: Duration(seconds: 1),
opacity: _isVisible ? 1.0 : 0.0,
child: Text(
'Fade Animation',
style: TextStyle(fontSize: 24.0),
),
),
SizedBox(height: 20),
AnimatedContainer(
duration: Duration(seconds: 1),
height: _isVisible ? 100.0 : 0.0,
curve: Curves.easeInOut,
child: SlideTransition(
position: Tween<Offset>(
begin: Offset(0, -1),
end: Offset.zero,
).animate(CurvedAnimation(
parent: ModalRoute.of(context).animation!,
curve: Curves.easeInOut,
)),
child: Container(
color: Colors.blue,
child: Center(
child: Text(
'Slide Animation',
style: TextStyle(fontSize: 24.0, color: Colors.white),
),
),
),
),
),
),
```

```
],
),
),
floatingActionButton: FloatingActionButton(
onPressed: toggleVisibility,
child: Icon(Icons.play_arrow),
),
);
}
}
```

9. a) Fetch data from a REST API.

To fetch data from a REST API, you typically need to make an HTTP request to the API endpoint. Below, I'll provide an example of how you can do this in Python using the popular requests library

Program

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

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

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

class _MyAppState extends State<MyApp> {
List<dynamic> _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) {
setState(() {
_data = json.decode(response.body);
});
} else {
throw Exception('Failed to load data');
}
}

@override
Widget build(BuildContext context) {
```

```
return MaterialApp(
home: Scaffold(
appBar: AppBar(
title: Text('REST API Data'),
),
body: _data.isEmpty
? Center(
child: CircularProgressIndicator(),
)
: ListView.builder(
itemCount: _data.length,
itemBuilder: (BuildContext context, int index) {
return ListTile(
title: Text(_data[index]['title']),
subtitle: Text(_data[index]['description']),
);
},
),
),
);
}
}
```

b) Display the fetched data in a meaningful way in the UI.

In Flutter, you can display fetched data in various widgets depending on the type of data you have.

Program

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

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

class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
return MaterialApp(
title: 'Fetched Data Display',
home: MyHomePage(),
);
}
}

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

class _MyHomePageState extends State<MyHomePage> {
List<String> fetchedData = []; // Assume you fetched a list of strings

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

```
fetchData(); // Function to fetch data
}

void fetchData() {
// Assume fetching data from an API or database
// For example, you can populate fetchedData with dummy data
setState(() {
fetchedData = [
"Item 1",
"Item 2",
"Item 3",
];
});
}

@override
Widget build(BuildContext context) {
return Scaffold(
appBar: AppBar(
title: Text('Fetched Data Display'),
),
body: ListView.builder(
itemCount: fetchedData.length,
itemBuilder: (context, index) {
return ListTile(
title: Text(fetchedData[index]),
);
},
),
);
}
}
```

10. a) Write unit tests for UI components.

Writing unit tests for UI components in Flutter typically involves testing widget behavior, rendering, and interactions.

Program

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

class MyButton extends StatelessWidget {
final String text;
final VoidCallback onPressed;

MyButton({required this.text, required this.onPressed});

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

}

b) Use Flutter's debugging tools to identify and fix issues.

Flutter provides a set of powerful debugging tools to identify and fix issues in your app. Here's a general process you can follow:

1. Inspect Widgets: Flutter's built-in widget inspector allows you to visualize the widget hierarchy and properties. You can enable this by pressing the "Toggle Widget Inspector" button in the debug toolbar, or by pressing Ctrl + Alt + Shift + D (on Windows/Linux) or Cmd + Option + Shift + D (on macOS) in your IDE.
2. Debug Paint: You can enable debug paint mode to see the layout boundaries and padding of each widget. This helps identify layout issues. Press the "Toggle Debug Paint" button in the debug toolbar or use the shortcut Ctrl + Alt + P (on Windows/Linux) or Cmd + Option + P (on macOS).
3. Debug Console: Use the debug console to print debug messages and inspect variable values. You can use print() statements to log messages to the console. Additionally, you can use the debugPrint() function for more controlled logging.
4. Flutter DevTools: Flutter DevTools is a suite of performance and debugging tools. You can run it by executing the flutter pub global run devtools command in your terminal and then navigating to the provided URL in your browser. DevTools provides insights into performance issues, memory usage, widget inspector, and more.
5. Flutter Inspector: This tool, available within the DevTools suite, provides a detailed view of your widget hierarchy and their properties. You can inspect widgets, view their constraints, and check for layout issues.
6. Hot Reload and Hot Restart: Utilize hot reload (R in the terminal or IDE) and hot restart (Shift + R in the terminal or IDE) to quickly see the effect of code changes. Hot reload updates the UI state while preserving the app's state, whereas hot restart resets the app's state.
7. Debugging in IDE: If you're using an IDE like VS Code or Android Studio, take advantage of their debugging capabilities. Set breakpoints, step through code, and inspect variables to identify and fix issues.

By leveraging these debugging tools effectively, you can efficiently identify and resolve issues in your Flutter app.