

Name:Sujal.S.Tekwani

Class:D15B

Roll No:59

MAD LAB-1

Aim: Installation and Configuration of Flutter Environment with Android Studio.

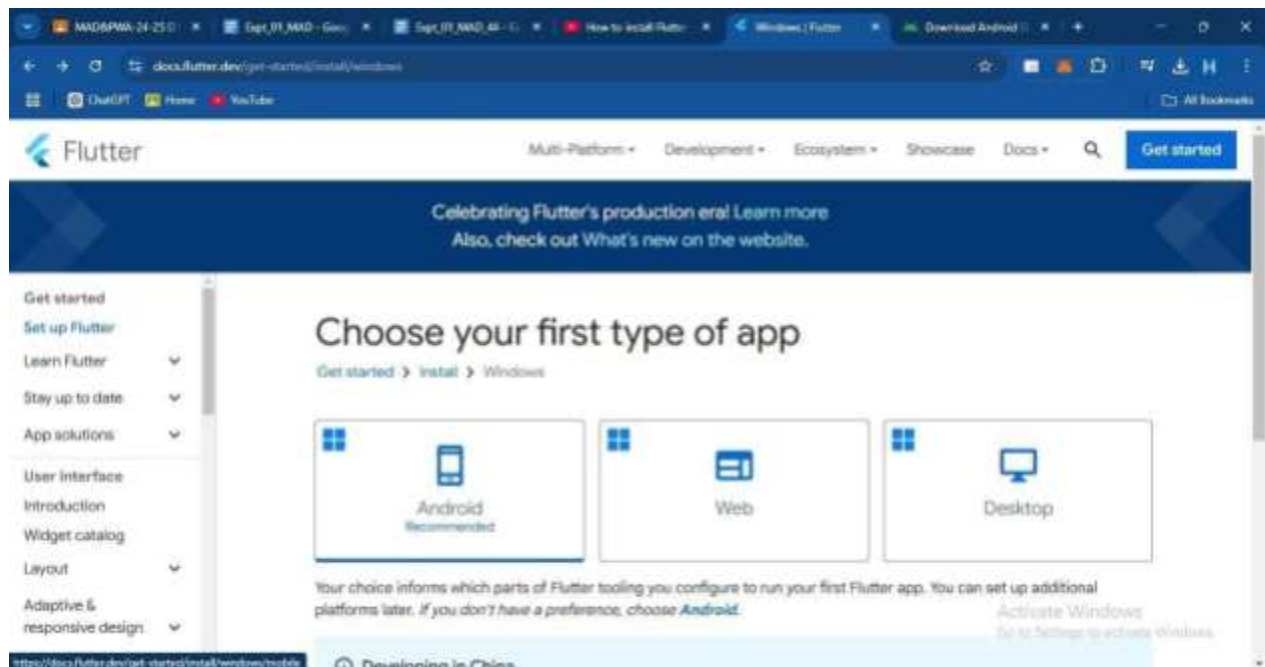
Theory:

Flutter is an open-source framework for building cross-platform mobile, web, and desktop applications from a single codebase. To start developing with Flutter, the following tools need to be installed:

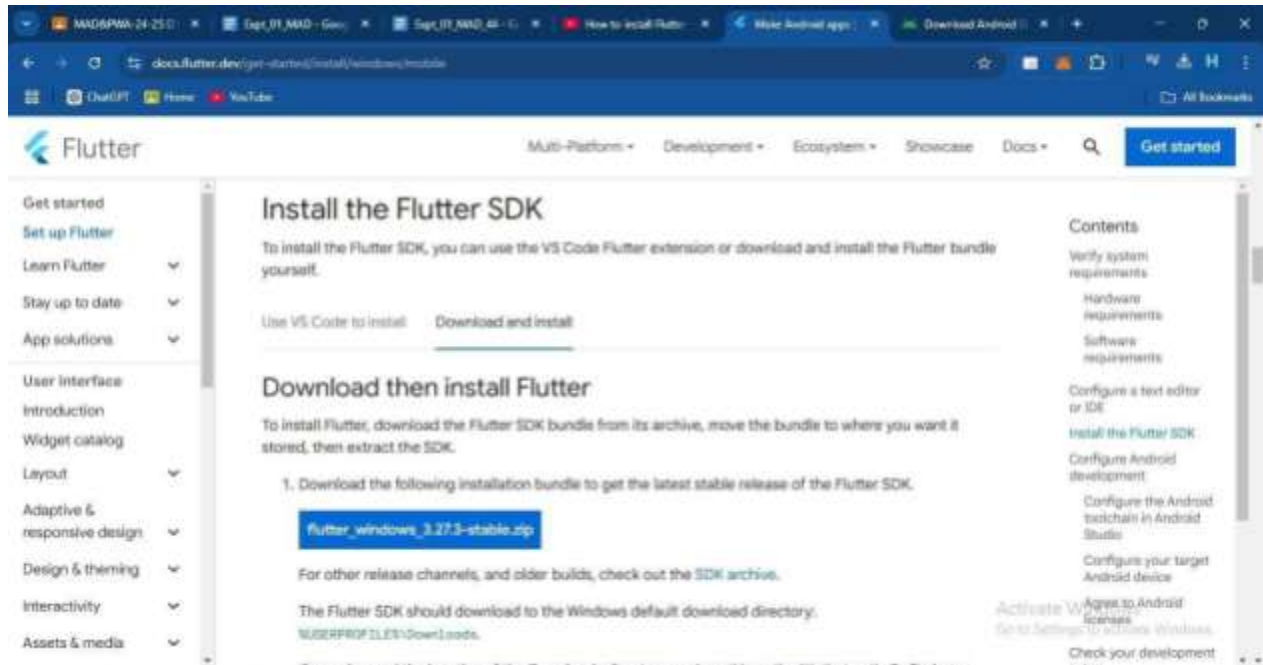
1. Flutter SDK - Contains necessary tools and libraries for building apps.
2. Dart SDK - Flutter uses Dart as its programming language.
3. Android Studio or Visual Studio Code - Code editors with plugins for Flutter development.
4. Xcode (for macOS users) - For iOS development and testing.
5. Set up Emulators or Physical Devices for testing apps.

Install the Flutter SDK

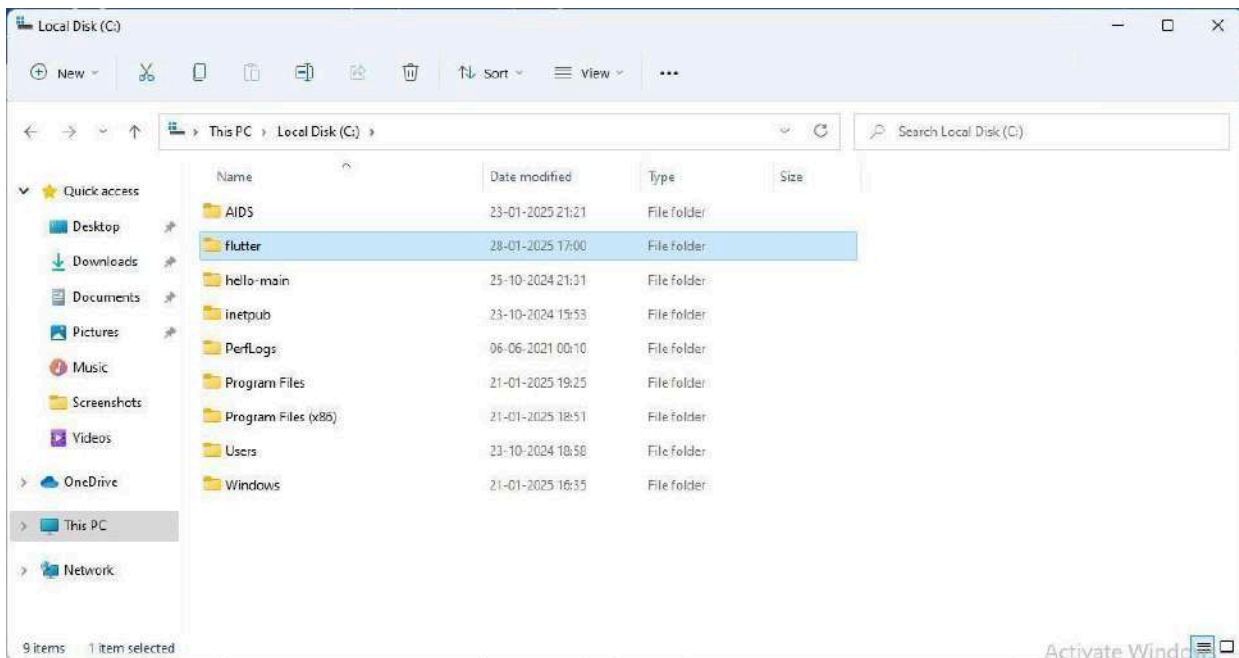
Step 1: Download the installation bundle of the Flutter Software Development Kit for windows. To download Flutter SDK, Go to its official website <https://docs.flutter.dev/get-started/install> , you will get the following screen.



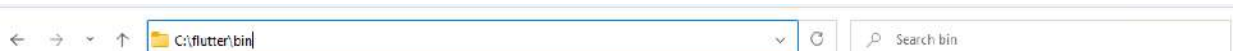
Step 2: Next, to download the latest Flutter SDK, click on the Windows icon. Here, you will find the download link for SDK.



Step 3: When your download is complete, extract the zip file and place it in the desired installation folder or location, for example, C: /flutter.

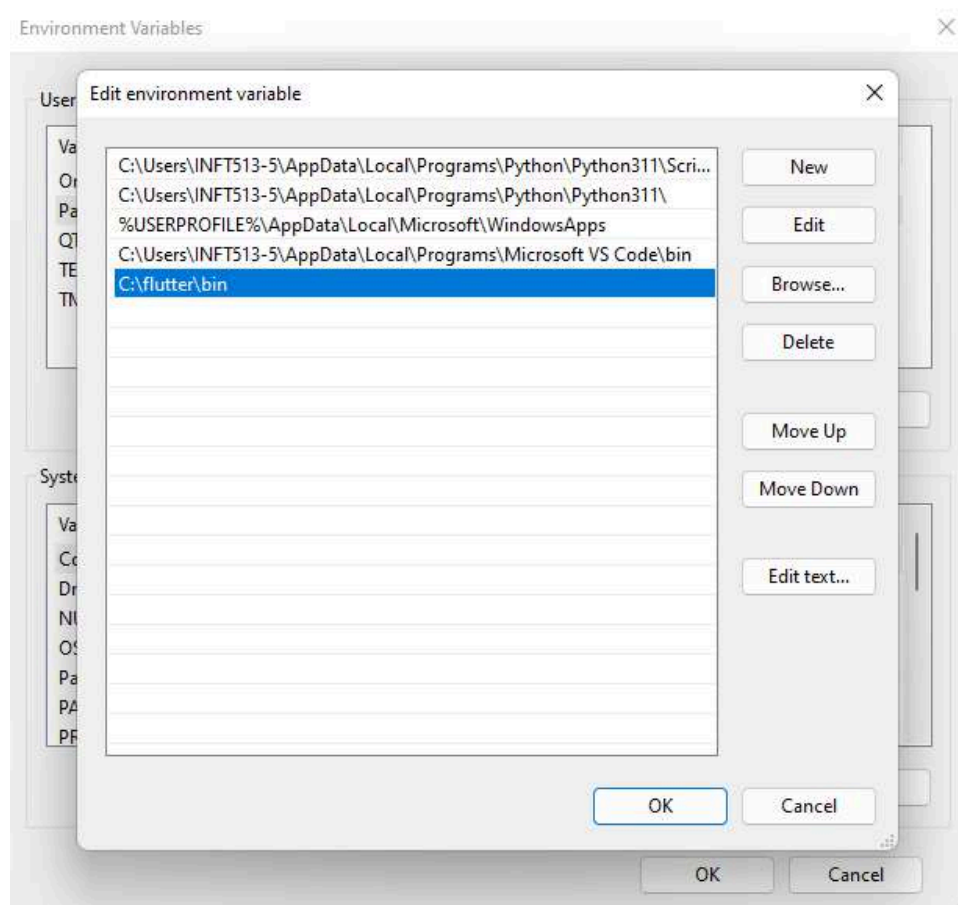


Step 4: To run the Flutter command in regular windows console, you need to update the system path to include the flutter bin directory. The following steps are required to do this:



Step 4.1: Go to MyComputer properties -> advanced tab -> environment variables. You will get the following screen.

Step 4.2: Now, select path -> click on edit. The following screen appears



Step 4.3: In the above window, click on New->write path of Flutter bin folder in variable value -> ok -> ok -> ok.

Step 5: Now, run the \$ flutter command in the command prompt.

```
C:\Users\INFT513-5>flutter
Manage your Flutter app development.
```

Common commands:

```
flutter create <output directory>
Create a new Flutter project in the specified directory.
```

Now, run the \$ flutter doctor command. This command checks for all the requirements of Flutter app development and displays a report of the status of your Flutter installation.

```
Command Prompt - flutter - flutter doctor

C:\Users\INFT513-5>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 3.27.3, on Microsoft Windows [Version 10.0.22000.2538], locale en-IN)
[✓] Windows Version (Installed version of Windows is version 10 or higher)
[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/to/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 Windows apps
    X Visual Studio not installed; this is necessary to develop Windows apps.
      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)
[✓] VS Code (version 1.94.2)
[✓] Connected device (3 available)
[✓] Network resources

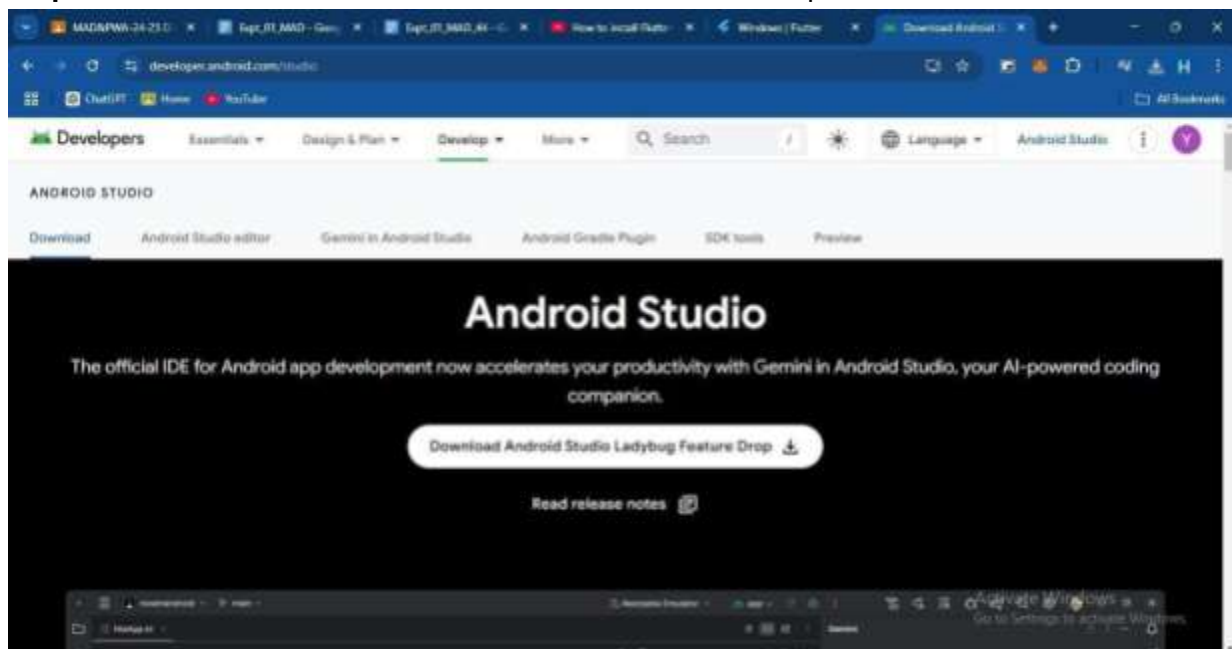
! Doctor found issues in 3 categories.

C:\Users\INFT513-5>
```

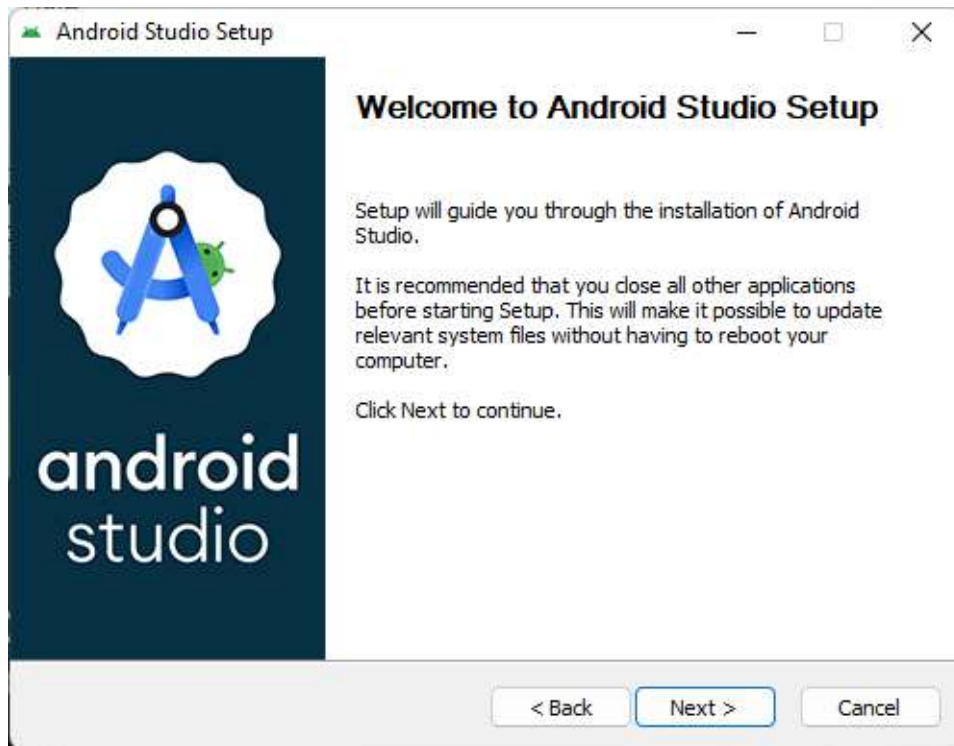
Step 6: When you run the above command, it will analyze the system and show its report, as shown in the below image. Here, you will find the details of all missing tools, which required to run Flutter as well as the development tools that are available but not connected with the device.

Step 7: Install the Android SDK. If the flutter doctor command does not find the Android SDK tool in your system, then you need first to install the Android Studio IDE. To install Android Studio IDE, do the following steps.

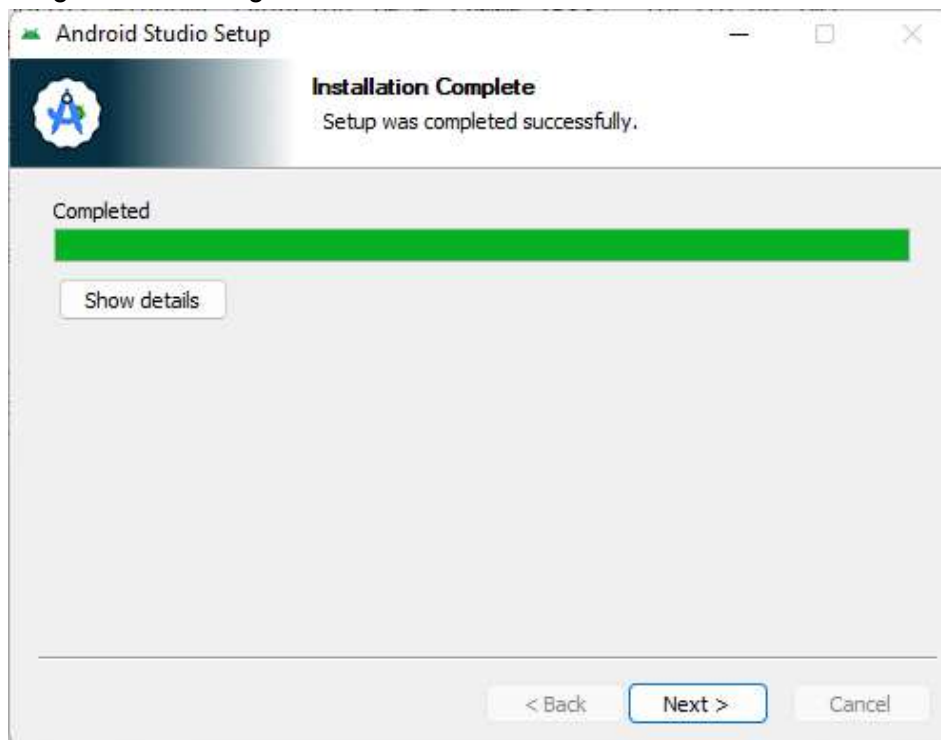
Step 7.1: Download the latest Android Studio executable or zip file from the official site.



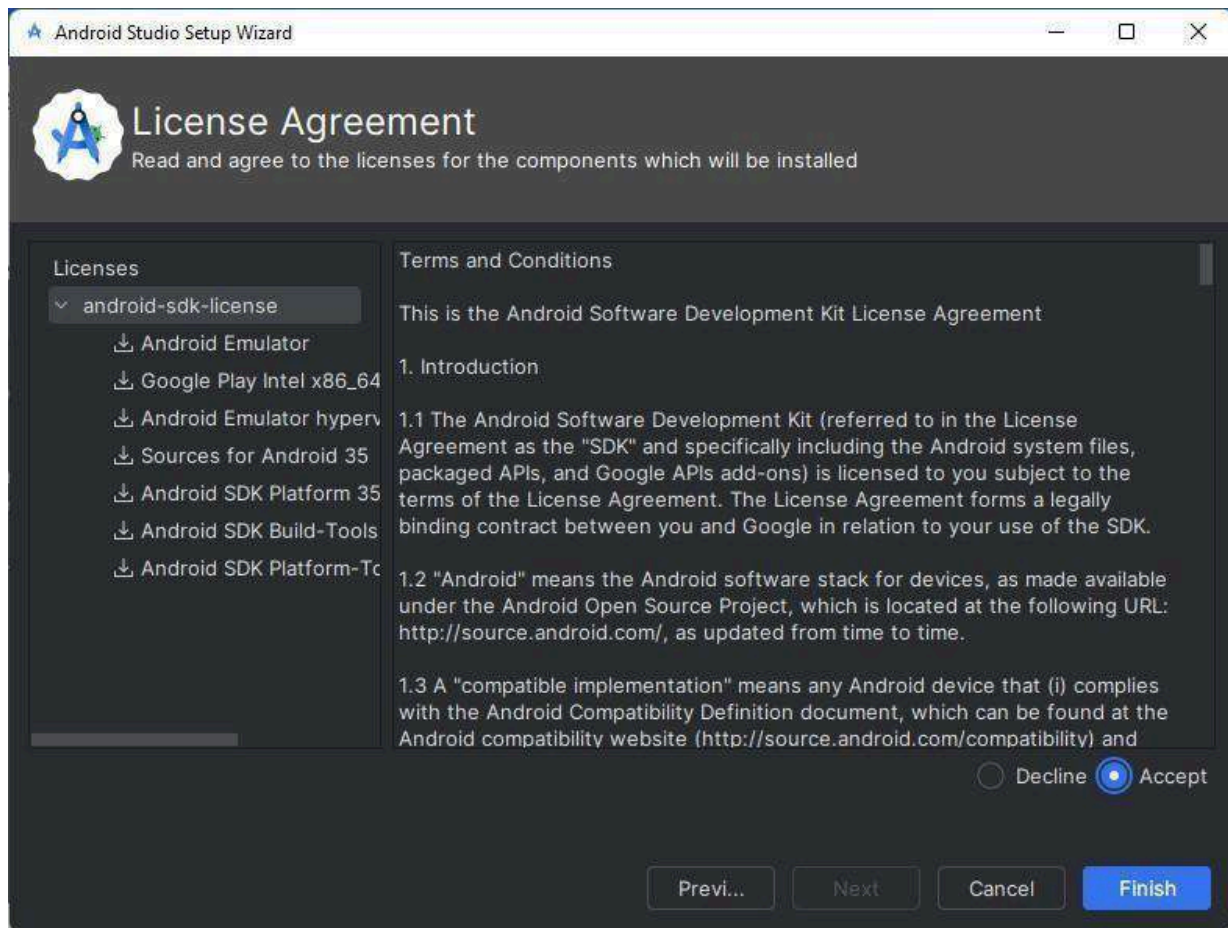
Step 7.2: When the download is complete, open the .exe file and run it. You will get the following dialog box.

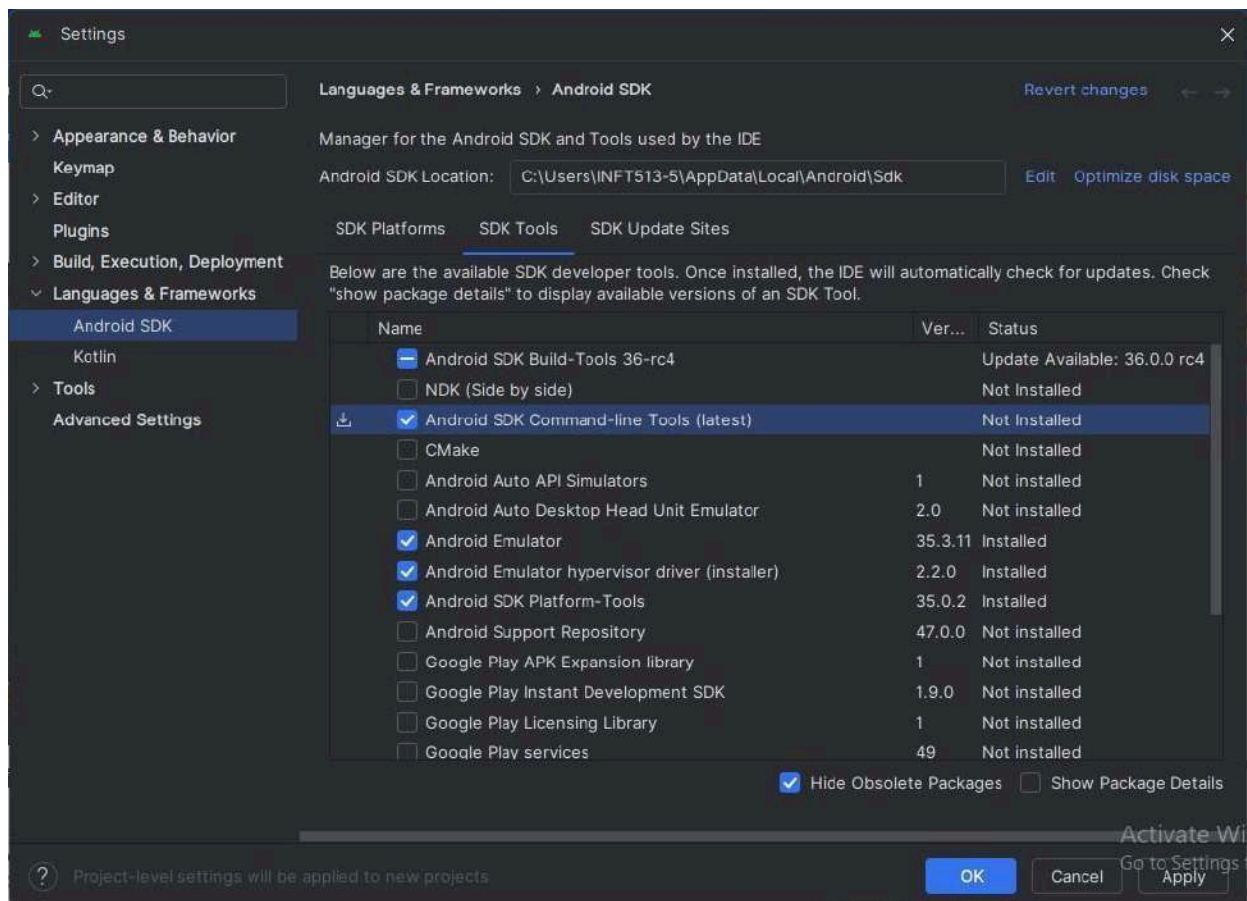


Step 7.3: Follow the steps of the installation wizard. Once the installation wizard completes, you will get the following screen.



Step 7.4: In the above screen, click Next-> Finish. Once the Finish button is clicked, you need to choose the 'Don't import Settings option' and click OK. It will start the Android Studio





Step 7.5: run the \$ flutter doctor command and Run flutter doctor --android-licenses command.

```
C:\Users\INFT513-5>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 3.27.3, on Microsoft Windows [Version 10.0.22000.2538], locale en-IN)
[✓] Windows Version (Installed version of Windows is version 10 or higher)
[!] Android toolchain - develop for Android devices (Android SDK version 35.0.1)
    ! Some Android licenses not accepted. To resolve this, run: flutter doctor --android-licenses
[✓] Chrome - develop for the web
[X] Visual Studio - develop Windows apps
    X Visual Studio not installed; this is necessary to develop Windows apps.
      Download at https://visualstudio.microsoft.com/downloads/.
      Please install the "Desktop development with C++" workload, including all of its default components
[✓] Android Studio (version 2024.2)
[✓] VS Code (version 1.94.2)
[✓] Connected device (3 available)
[✓] Network resources

! Doctor found issues in 2 categories.
```

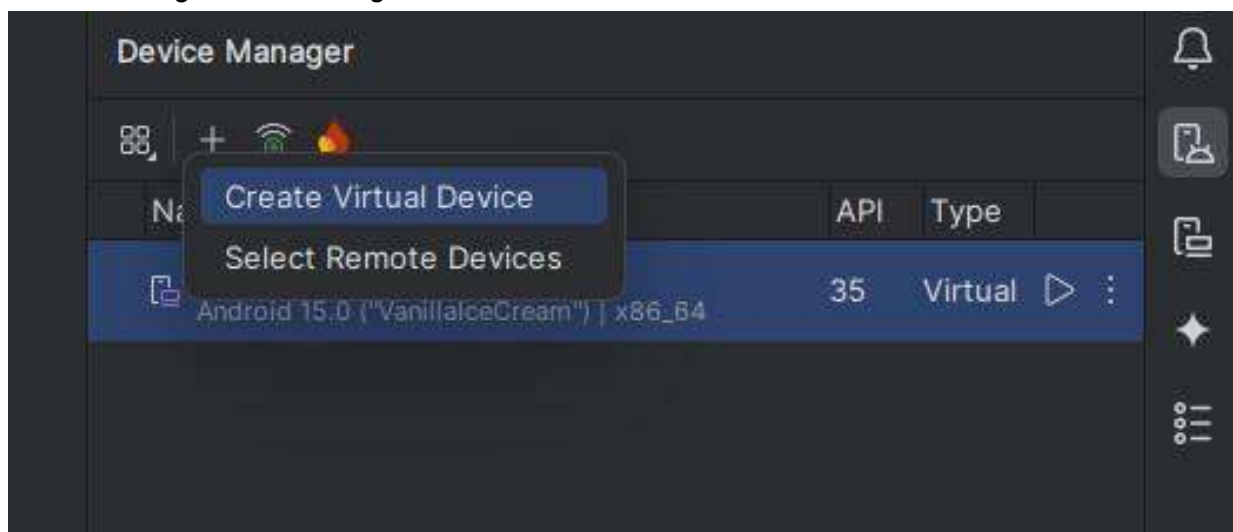
```
C:\Users\INFT513-5>flutter doctor --android-licenses
```

```
C:\Users\INFT513-5>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 3.27.3, on Microsoft Windows [Version 10.0.22000.2538], locale en-IN)
[✓] Windows Version (Installed version of Windows is version 10 or higher)
[✓] Android toolchain - develop for Android devices (Android SDK version 35.0.1)
[✓] Chrome - develop for the web
[X] Visual Studio - develop Windows apps
    X Visual Studio not installed; this is necessary to develop Windows apps.
      Download at https://visualstudio.microsoft.com/downloads/.
      Please install the "Desktop development with C++" workload, including all of its default components
[✓] Android Studio (version 2024.2)
[✓] VS Code (version 1.94.2)
[✓] Connected device (3 available)
[✓] Network resources

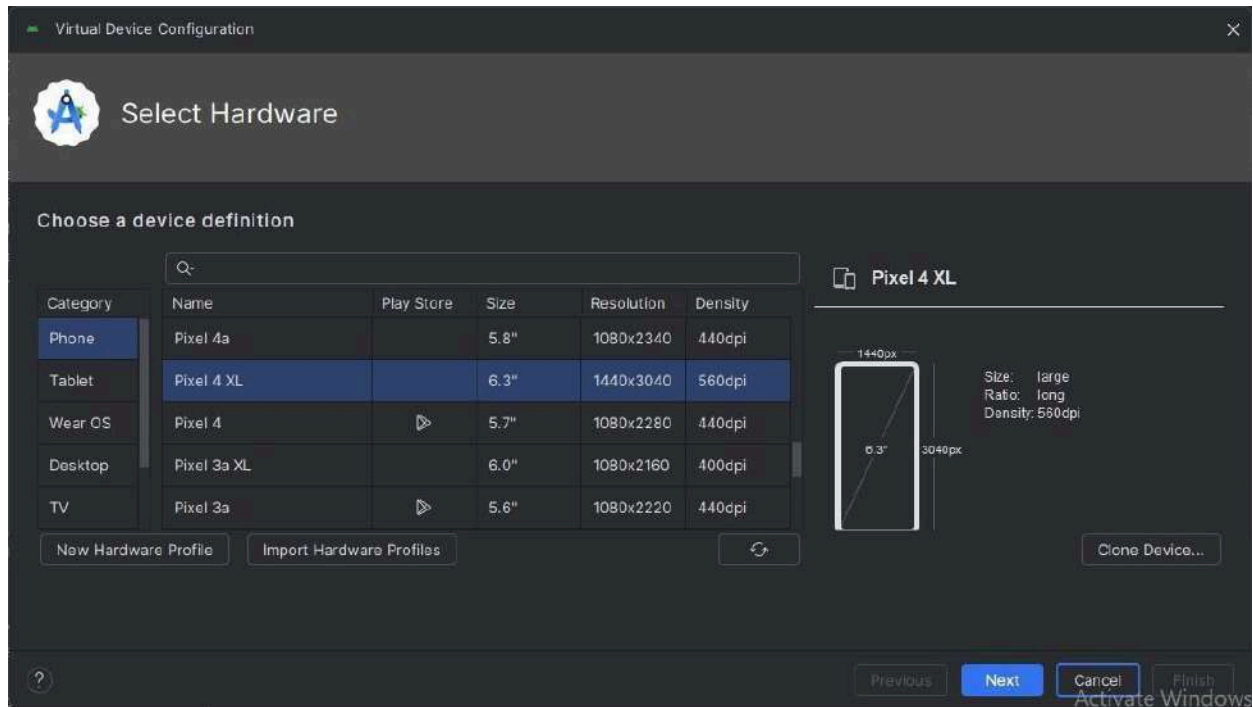
! Doctor found issues in 1 category.
```

Step 8: Next, you need to set up an Android emulator. It is responsible for running and testing the Flutter application.

Step 8.1: To set an Android emulator, go to Android Studio > Tools > Android > AVD Manager and select Create Virtual Device. Or, go to Help->Find Action->Type Emulator in the search box. You will get the following screen.

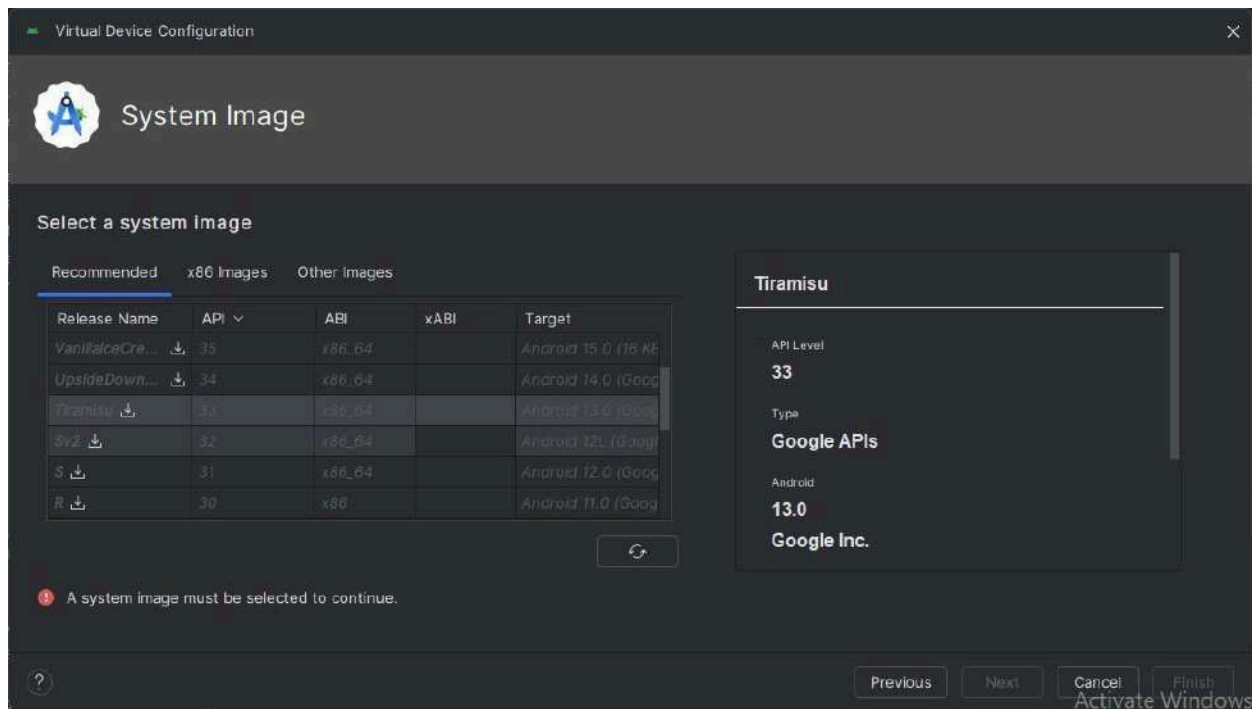


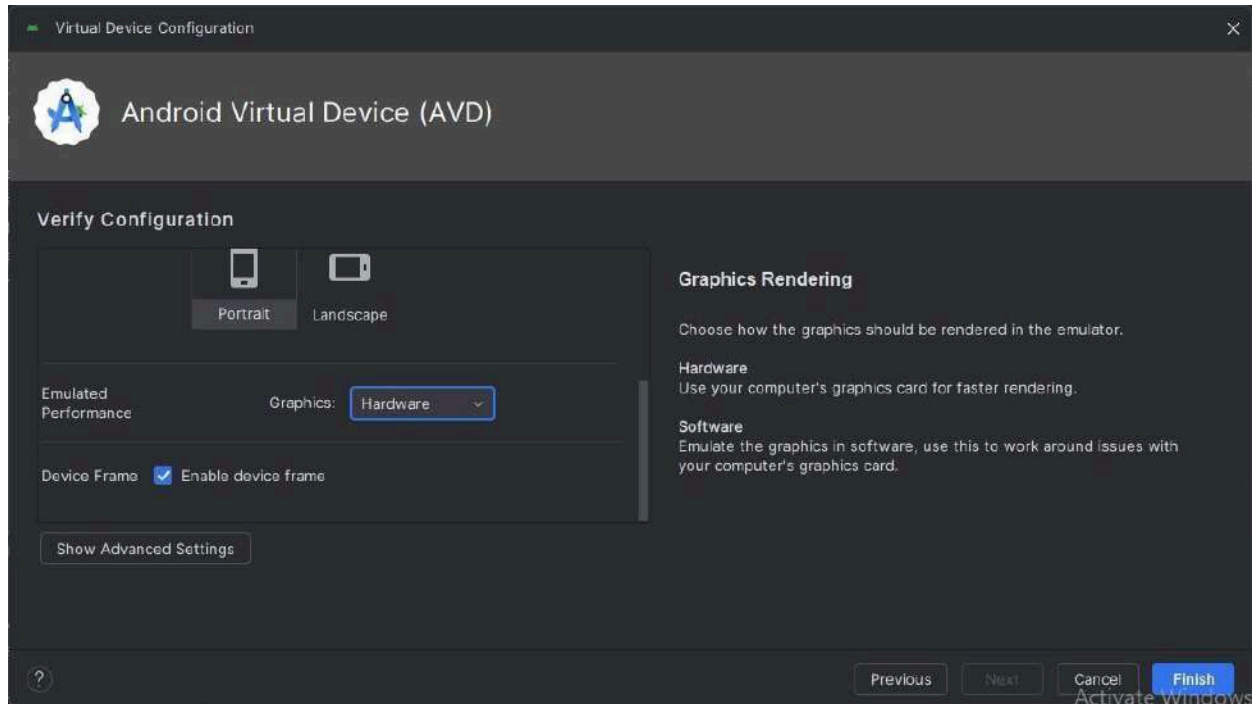
Step 8.2: Choose your device definition and click on Next.



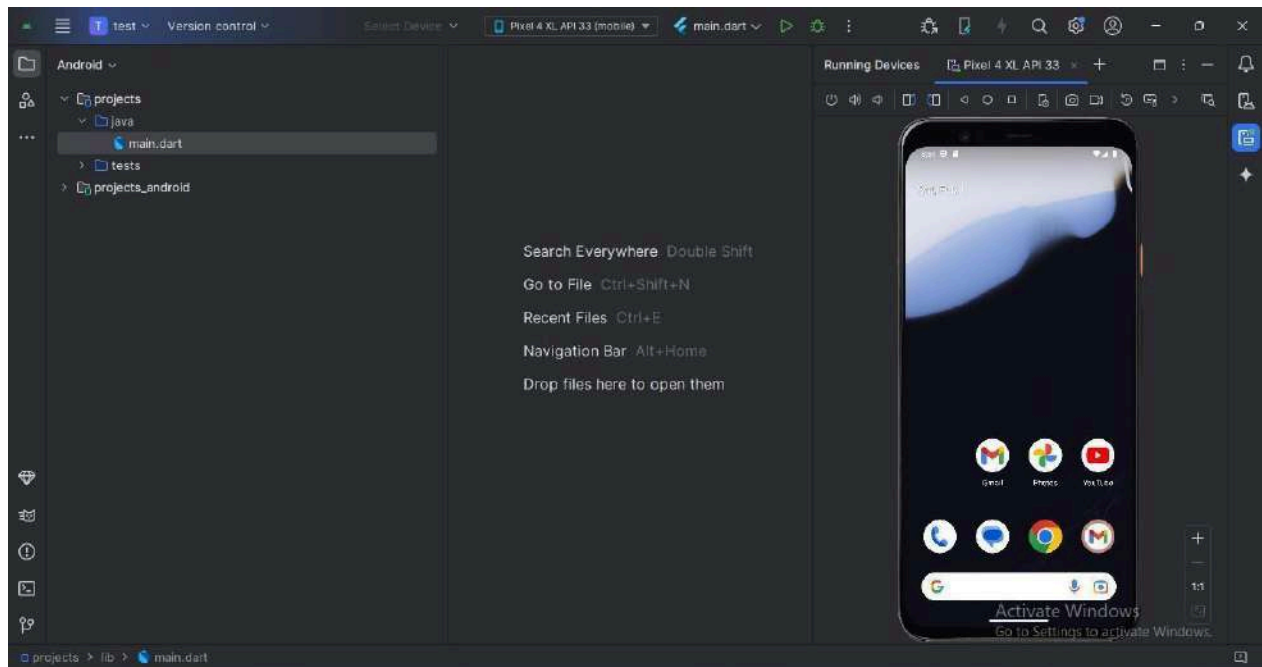
Step 8.3: Select the system image for the latest Android version and click on Next.

Step 8.4: Now, verify the all AVD configuration. If it is correct, click on Finish. The following screen appears.



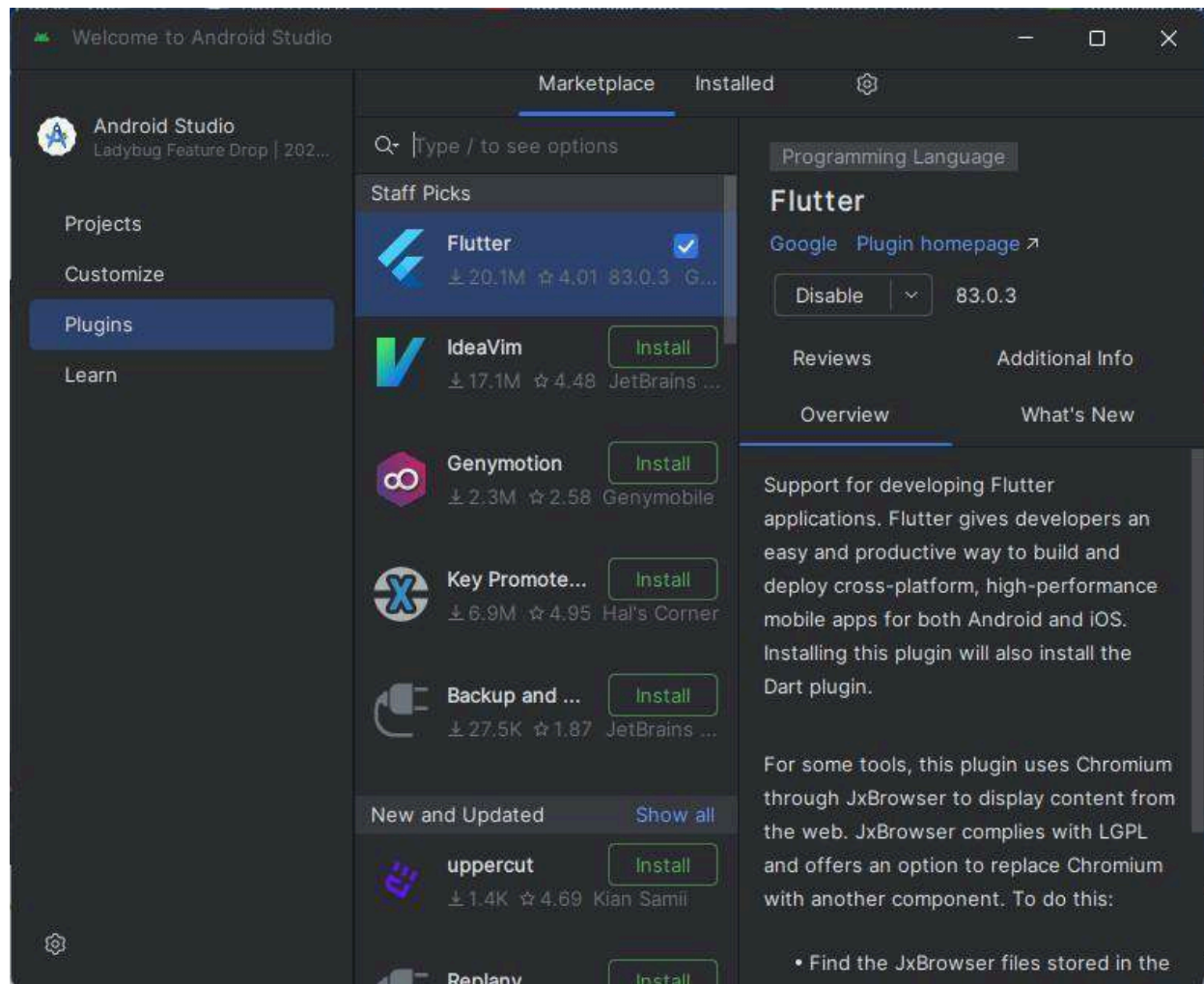


Step 8.5: Last, click on the icon pointed into the red color rectangle. The Android emulator displayed as below screen.



Step 9: Now, install Flutter and Dart plugin for building Flutter application in Android Studio. These plugins provide a template to create a Flutter application, give an option to run and debug Flutter application in the Android Studio itself. Do the following steps to install these plugins.

Step 9.1: Open the Android Studio and then go to File->Settings->Plugins.



Step 9.2: Now, search the Flutter plugin. If found, select Flutter plugin and click install. When you click on install, it will ask you to install Dart plugin as below screen. Click yes to proceed.

Step 9.3: Restart the Android Studio.

Conclusion:

Successfully installing and configuring the Flutter environment ensures that developers can begin building cross-platform applications. The setup includes installing essential tools and dependencies, allowing the smooth creation, testing, and deployment of applications with Flutter.

Name: Sujal.S.Tekwani

Class: D15B

Roll No: 59

MAD LAB-2

Aim: To design flutter UI using common widgets

Code :

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

```
class LoginScreen extends StatefulWidget {
```

```
  @override
```

```
  _LoginScreenState createState() => _LoginScreenState();
```

```
}
```

```
class _LoginScreenState extends State<LoginScreen> {
```

```
  final TextEditingController emailController = TextEditingController();
```

```
  final TextEditingController passwordController = TextEditingController();
```

```
  @override
```

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

```
    return Scaffold(
```

```
      backgroundColor: Colors.black, // Dark background
```

```
      body: Center(
```

```
        child: Padding(
```

```
          padding: EdgeInsets.symmetric(horizontal: 20),
```

```
          child: Column(
```

```
            mainAxisAlignment: MainAxisAlignment.min,
```

```
            children: [
```

```
              Text(
```

```
                "Login",
```

```
                style: TextStyle(
```

```
                  fontSize: 32,
```

```
                  fontWeight: FontWeight.bold,
```

```
        color: Colors.limeAccent,
    ),
),
    SizedBox(height: 20),
    _buildTextField(
        controller: emailController,
        label: "Email",
        icon: Icons.email_outlined,
    ),
    SizedBox(height: 15),
    _buildTextField(
        controller: passwordController,
        label: "Password",
        icon: Icons.lock_outline,
        isPassword: true,
    ),
    SizedBox(height: 10),
    Align(
        alignment: Alignment.centerRight,
        child: TextButton(
            onPressed: () {},
            child: Text(
                "Forget Password",
                style: TextStyle(color: Colors.limeAccent),
            ),
        ),
    ),
    SizedBox(height: 10),
    ElevatedButton(
        onPressed: () {},
        style: ElevatedButton.styleFrom(
            backgroundColor: Colors.black,
```

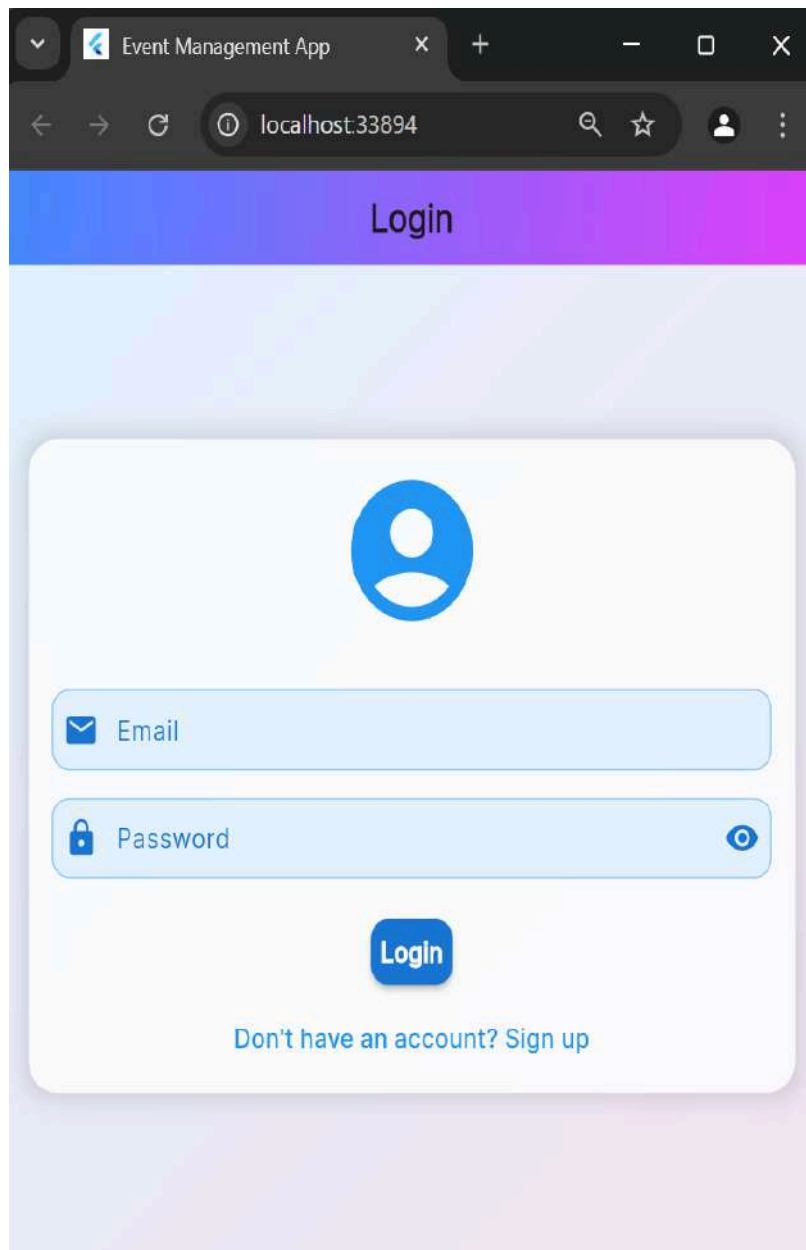
```
padding: EdgeInsets.symmetric(vertical: 14, horizontal: 100),
shape: RoundedRectangleBorder(
  borderRadius: BorderRadius.circular(8),
),
),
child: Text(
  "Login",
  style: TextStyle(color: Colors.limeAccent),
),
),
 SizedBox(height: 20),
  TextButton(
    onPressed: () {},
    child: Text.rich(
      TextSpan(
        text: "Create a New Account ? ",
        style: TextStyle(color: Colors.limeAccent),
        children: [
          TextSpan(
            text: "Sign Up",
            style: TextStyle(
              fontWeight: FontWeight.bold,
              color: Colors.limeAccent,
            ),
          ),
        ],
      ),
    ),
  ),
],
),
),
```



```
    ),  
  );  
}
```

```
Widget _buildTextField({  
  required TextEditingController controller,  
  required String label,  
  required IconData icon,  
  bool isPassword = false,  
}) {  
  return TextField(  
    controller: controller,  
    obscureText: isPassword,  
    decoration: InputDecoration(  
      filled: true,  
      fillColor: Colors.limeAccent,  
      prefixIcon: Icon(icon, color: Colors.black),  
      labelText: label,  
      labelStyle: TextStyle(color: Colors.black),  
      border: OutlineInputBorder(  
        borderRadius: BorderRadius.circular(12),  
        borderSide: BorderSide.none,  
      ),  
    ),  
  ),  
);  
}
```

Output:



Name:Sujal.S.Tekwani

Class:D15B

Roll No:59

MAD LAB-3

Aim: To include icons, images and fonts in App.

Code :

Sign Up Page:

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

```
class SignUpScreen extends StatefulWidget {  
  @override  
  _SignUpScreenState createState() => _SignUpScreenState();  
}
```

```
class _SignUpScreenState extends State<SignUpScreen> {  
  final TextEditingController nameController = TextEditingController();  
  final TextEditingController emailController = TextEditingController();  
  final TextEditingController passwordController = TextEditingController();
```

```
  @override
```

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

```
    return Scaffold(  
      backgroundColor: Colors.black, // Dark background
```

```
      body: Center(  
        child: Padding(  
          padding: EdgeInsets.symmetric(horizontal: 20),  
          child: Column(  
            mainAxisAlignment: MainAxisAlignment.min,  
            children: [  
              Text(  
                "Sign Up",  
                style: TextStyle(  
                  color: Colors.white, // White text
```

```
        fontSize: 32,
        fontWeight: FontWeight.bold,
        color: Colors.limeAccent,
      ),
    ),
    SizedBox(height: 20),
    _buildTextField(
      controller: nameController,
      label: "Name",
      icon: Icons.person_outline,
    ),
    SizedBox(height: 15),
    _buildTextField(
      controller: emailController,
      label: "Email",
      icon: Icons.email_outlined,
    ),
    SizedBox(height: 15),
    _buildTextField(
      controller: passwordController,
      label: "Password",
      icon: Icons.lock_outline,
      isPassword: true,
    ),
    SizedBox(height: 15),
    ElevatedButton(
      onPressed: () {},
      style: ElevatedButton.styleFrom(
        backgroundColor: Colors.black,
        padding: EdgeInsets.symmetric(vertical: 14, horizontal: 100),
        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(8),
```

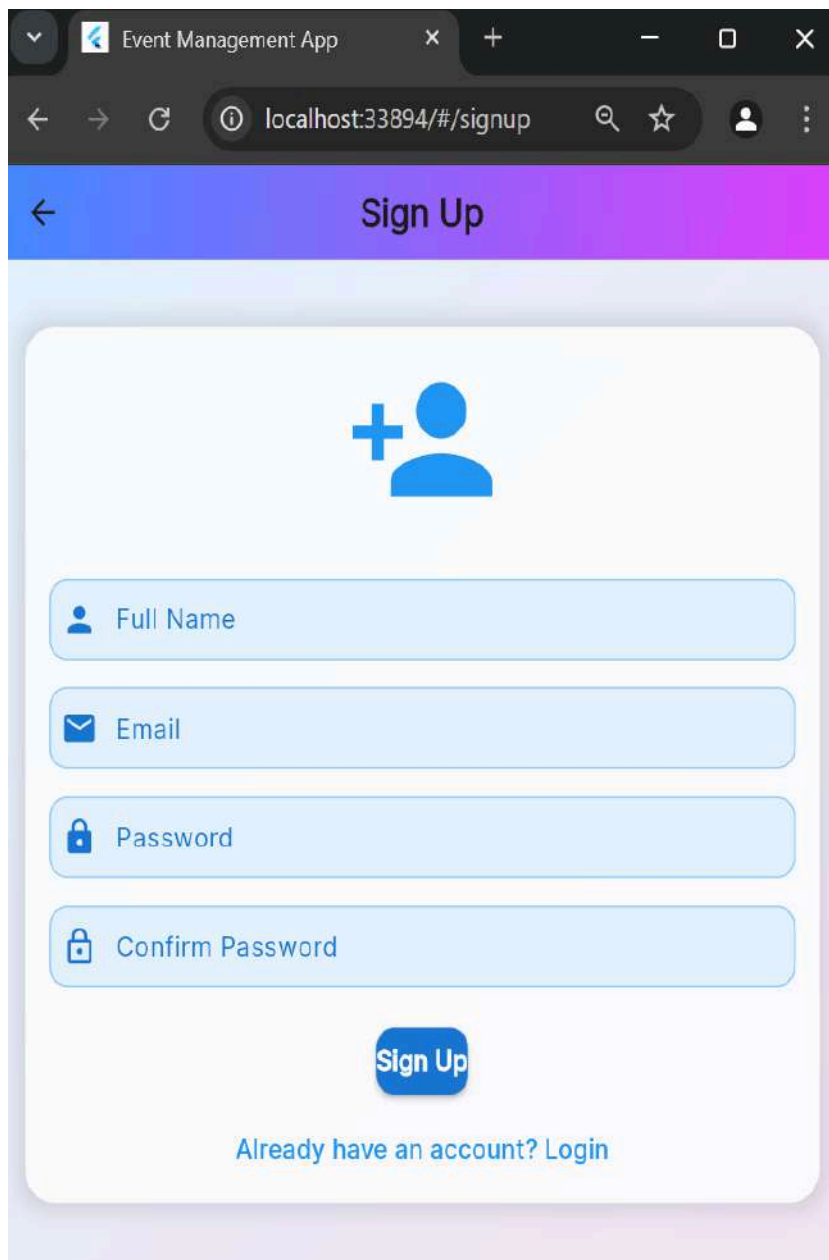
```

    ),
  ),
  child: Text(
    "Sign Up",
    style: TextStyle(color: Colors.limeAccent),
  ),
),
 SizedBox(height: 20),
  TextButton(
    onPressed: () {},
    child: Text.rich(
      TextSpan(
        text: "Already have an account? ",
        style: TextStyle(color: Colors.limeAccent),
        children: [
          TextSpan(
            text: "Login",
            style: TextStyle(
              fontWeight: FontWeight.bold,
              color: Colors.limeAccent,
            ),
          ),
        ],
      ),
    ),
  ),
],
);
}

```

```
Widget _buildTextField({
  required TextEditingController controller,
  required String label,
  required IconData icon,
  bool isPassword = false,
}) {
  return TextField(
    controller: controller,
    obscureText: isPassword,
    decoration: InputDecoration(
      filled: true,
      fillColor: Colors.limeAccent,
      prefixIcon: Icon(icon, color: Colors.black),
      labelText: label,
      labelStyle: TextStyle(color: Colors.black),
      border: OutlineInputBorder(
        borderRadius: BorderRadius.circular(12),
        borderSide: BorderSide.none,
      ),
    ),
  );
}
```


Output:





The screenshot displays a web browser window with the title 'Event Management App' and the address bar showing 'localhost:33894/#/signup'. The page features a purple header with a back arrow and the text 'Sign Up'. Below the header is a light blue rounded rectangle containing a blue icon of a person with a plus sign. There are four input fields, each with a blue icon on the left: 'Full Name' (person icon), 'Email' (envelope icon), 'Password' (lock icon), and 'Confirm Password' (lock icon). A blue 'Sign Up' button is positioned below the fields. At the bottom, there is a link that says 'Already have an account? Login'.


Event Management App


localhost:33894/#/signup


← Sign Up

+ 

 Full Name

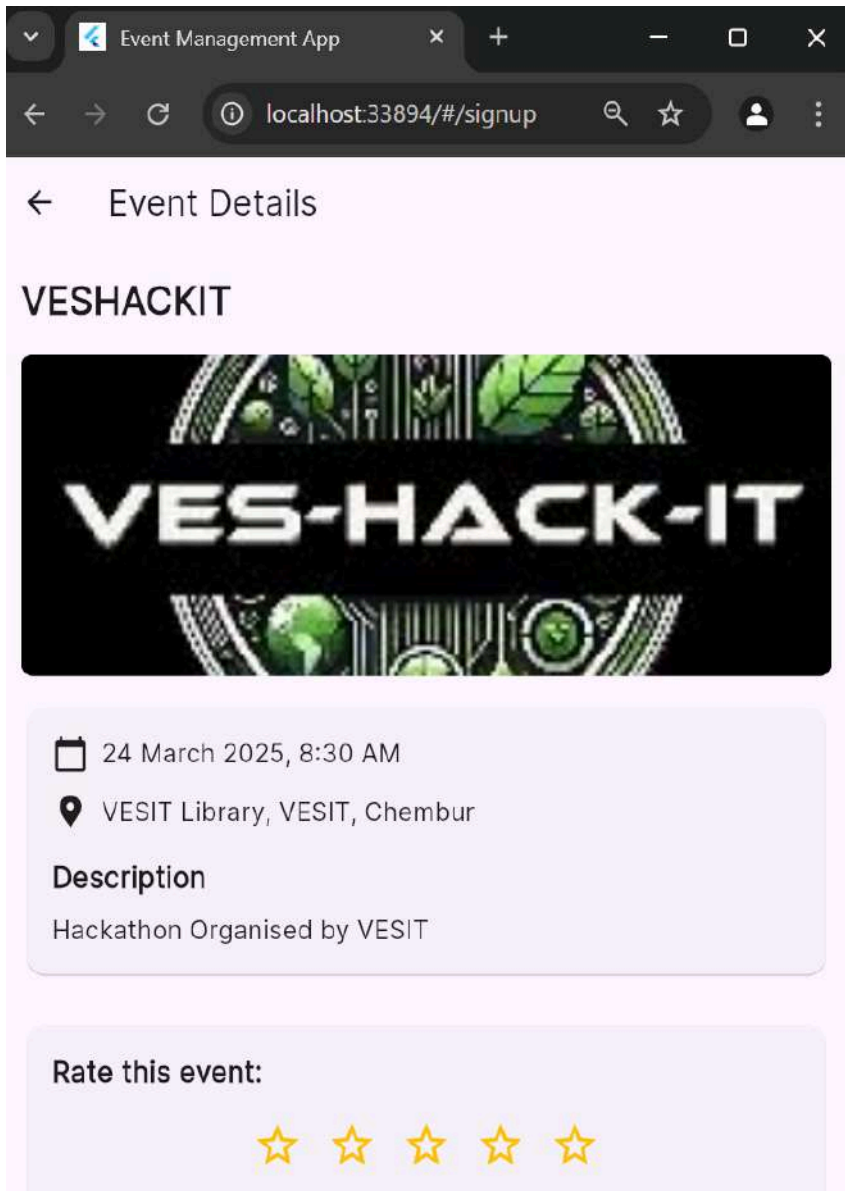
 Email

 Password

 Confirm Password

Sign Up

Already have an account? [Login](#)



Name:Sujal.S.Tekwani

Class:D15B

Roll No:59

MAD LAB-4

Aim: To create an interactive Form using form widget

Code :

Form to Create an Event:

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

```
class CreateEventScreen extends StatefulWidget {  
  @override  
  _CreateEventScreenState createState() => _CreateEventScreenState();  
}
```

```
class _CreateEventScreenState extends State<CreateEventScreen> {  
  bool isInPerson = false;
```

```
  @override
```

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

```
    return Scaffold(  
      backgroundColor: Colors.black,
```

```
      appBar: AppBar(  
        title: Text(  
          "Create Event",  
          style: TextStyle(color: Colors.limeAccent, fontSize: 24),  
        ),  
      ),  
      backgroundColor: Colors.black,  
      elevation: 0,  
    ),  
    body: Padding(  
      padding: EdgeInsets.all(16.0),  
      child: Column(  
        mainAxisAlignment: MainAxisAlignment.spaceBetween,  
        children: [  
          Text("Name"),  
          TextFormField(),  
          Text("Email"),  
          TextFormField(),  
          Text("Phone"),  
          TextFormField(),  
          Text("Address"),  
          TextFormField(),  
          Text("Event Type"),  
          TextFormField(),  
          Text("Date"),  
          TextFormField(),  
          Text("Time"),  
          TextFormField(),  
          Text("Location"),  
          TextFormField(),  
          Text("Notes"),  
          TextFormField(),  
          Text("Create Event"),  
          ElevatedButton(),  
        ],  
      ),  
    ),  
  ),  
}
```

```

children: [
  // Image Upload Section
  Container(
    height: 150,
    width: double.infinity,
    decoration: BoxDecoration(
      color: Colors.limeAccent,
      borderRadius: BorderRadius.circular(10),
    ),
    child: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          Icon(Icons.add_a_photo, size: 40, color: Colors.black),
          SizedBox(height: 5),
          Text(
            "Add Event Image",
            style: TextStyle(color: Colors.black),
          ),
        ],
      ),
    ),
  ),
  SizedBox(height: 20),

  // Input Fields
  _buildTextField("Event Name", Icons.event),
  _buildTextField("Description", Icons.description),
  _buildTextField("Location", Icons.location_on),
  _buildTextField("Date & Time", Icons.calendar_today),
  _buildTextField("Guests", Icons.people),
  _buildTextField("Sponsors", Icons.attach_money),

```

```
SizedBox(height: 15),
```

```
// In-Person Event Toggle
```

```
Container(  
  padding: EdgeInsets.symmetric(vertical: 10, horizontal: 10),  
  decoration: BoxDecoration(  
    color: Colors.black,  
    borderRadius: BorderRadius.circular(5),  
  ),  
  child: Row(  
    mainAxisAlignment: MainAxisAlignment.spaceBetween,  
    children: [  
      Text(  
        "In Person Event",  
        style: TextStyle(color: Colors.limeAccent, fontSize: 16),  
      ),  
      Switch(  
        value: isInPerson,  
        activeColor: Colors.limeAccent,  
        onChanged: (value) {  
          setState(() {  
            isInPerson = value;  
          });  
        },  
      ),  
    ],  
  ),  
)
```

```
SizedBox(height: 15),
```

```

// Create Event Button
ElevatedButton(
  onPressed: () {},
  style: ElevatedButton.styleFrom(
    backgroundColor: Colors.limeAccent,
    padding: EdgeInsets.symmetric(vertical: 14, horizontal: 60),
    shape: RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(8),
    ),
  ),
  child: Text(
    "Create New Event",
    style: TextStyle(color: Colors.black, fontSize: 16),
  ),
),
],
),
),
);
}

```

```

Widget _buildTextField(String label, IconData icon) {
  return Padding(
    padding: EdgeInsets.symmetric(vertical: 5),
    child: TextField(
      decoration: InputDecoration(
        filled: true,
        fillColor: Colors.limeAccent,
        prefixIcon: Icon(icon, color: Colors.black),
        labelText: label,
        labelStyle: TextStyle(color: Colors.black),
        border:

```


Output:

The screenshot displays a web browser window with the title 'Event Management App'. The address bar shows 'localhost:33894/#/signup'. The page has a purple header bar with a back arrow and the text 'Create Event'. Below the header, the section is titled 'Create New Event' with the instruction 'Fill in the event details below'. A large grey box contains a camera icon and the text 'Add Event Image'. Below this are several input fields: 'Event Name', 'Description', 'Location', 'Select Date' (with a right arrow), 'Guests', 'Sponsors', and 'Select Time' (with a right arrow). A toggle switch is visible to the right of the 'Sponsors' field. At the bottom, there is a large purple button labeled 'Create Event'.

Event Management App

localhost:33894/#/signup

Create Event

Create New Event

Fill in the event details below

Add Event Image

Event Name

Description

Location

Select Date

Guests

Sponsors

Select Time

Create Event

Name: Sujal.S.Tekwani

Class: D15B

Roll: 59

MAD Lab 5

Aim: Navigation, Routing, and Gestures in Flutter

Introduction

Navigation, routing, and gestures play a crucial role in creating an interactive user experience in Flutter applications. Navigation allows users to switch between different screens, routing ensures smooth transitions between pages, and gestures enable user interactions like tapping, swiping, and long pressing.

Routing and Navigation in Flutter

Flutter manages screen transitions through a **stack-based navigation system**, where pages are stacked on top of each other. This is handled using the **Navigator** widget, which allows adding (**push**) or removing (**pop**) screens dynamically.

Types of Navigation in Flutter

1. **Imperative Navigation (Navigator API)**
 - Uses **Navigator.push()** and **Navigator.pop()**
 - Follows a **Last In, First Out (LIFO)** structure
 - Best for **small-scale applications** with simple navigation
2. **Declarative Navigation (Go Router, Auto Route)**
 - Uses **URL-based navigation** for handling deep links
 - Ideal for **large applications** with complex navigation structures

Navigator and Routes in Flutter

The **Navigator widget** manages a **stack of screens**, where each screen is called a **Route**. It helps in **switching between screens smoothly** and maintains navigation history.

Code Implementation for Event Management App

Let's say your event management app has two screens:

1. **Home Screen** - Displays a list of events
2. **Event Details Screen** - Shows details of a selected event

Code for Navigation in the Event Management App

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

```
void main() {
```

```
  runApp(MaterialApp(home: HomeScreen()));
```

```
}
```

```
class HomeScreen extends StatelessWidget {  
  
  @override  
  
  Widget build(BuildContext context) {  
  
    return Scaffold(  
  
      appBar: AppBar(title: Text('Event Management')),  
  
      body: Center(  
  
        child: ElevatedButton(  
  
          onPressed: () {  
  
            // Navigating to EventDetailsScreen with event data  
  
            Navigator.push(  
  
              context,  
  
              MaterialPageRoute(  
  
                builder: (context) => EventDetailsScreen(eventName: "Tech Conference 2025"),  
  
              ),  
  
            );  
  
          },  
  
          child: Text('View Event Details'),  
  
        ),  
  
      ),  
  
    );  
  
  }  
  
}
```

```
class EventDetailsScreen extends StatelessWidget {  
  
  final String eventName;
```

```
EventDetailsScreen({required this.eventName});
```

```
@override
```

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

```
  return Scaffold(
```

```
    appBar: AppBar(title: Text('Event Details')),
```

```
    body: Center(
```

```
      child: Column(
```

```
        mainAxisAlignment: MainAxisAlignment.center,
```

```
        children: [
```

```
          Text('Event: $eventName', style: TextStyle(fontSize: 20, fontWeight: FontWeight.bold)),
```

```
          SizedBox(height: 20),
```

```
          ElevatedButton(
```

```
            onPressed: () {
```

```
              // Going back to HomeScreen
```

```
              Navigator.pop(context);
```

```
            },
```

```
            child: Text('Go Back'),
```

```
          ),
```

```
        ],
```

```
      ),
```

```
    ),
```

```
  );
```

```
}
```

```
}
```

2. Gesture Detection in Flutter

Flutter's GestureDetector widget captures various gestures like taps, double taps, long presses, and swipes. Below is an example of detecting different gestures.

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

void main() {
  runApp(MaterialApp(home: EventListScreen()));
}

class EventListScreen extends StatefulWidget {
  @override
  _EventListScreenState createState() => _EventListScreenState();
}

class _EventListScreenState extends State<EventListScreen> {
  String _message = "Interact with the event card";

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text("Event List")),
      body: GestureDetector(
        onTap: () {
          setState(() {
            _message = "Event selected!";
          });
        },
        onDoubleTap: () {
          setState(() {
            _message = "Event added to favorites!";
          });
        },
        onLongPress: () {
          setState(() {
            _message = "Event details opened!";
          });
          Navigator.push(
            context,
            MaterialPageRoute(
              builder: (context) => EventDetailsScreen(eventName: "Music Fest 2025"),
            ),
          );
        },
      ),
    );
  }
}
```

```

    );
  },
  onHorizontalDragEnd: (details) {
    setState(() {
      _message = "Swiped! Event removed from list.";
    });
  },
  child: Center(
    child: Container(
      padding: EdgeInsets.all(20),
      margin: EdgeInsets.symmetric(horizontal: 20, vertical: 50),
      decoration: BoxDecoration(
        color: Colors.blueAccent,
        borderRadius: BorderRadius.circular(15),
        boxShadow: [
          BoxShadow(color: Colors.black26, blurRadius: 5, spreadRadius: 2)
        ],
      ),
    child: Column(
      mainAxisAlignment: MainAxisAlignment.min,
      children: [
        Text(
          "Music Fest 2025",
          style: TextStyle(fontSize: 24, fontWeight: FontWeight.bold, color: Colors.white),
        ),
        SizedBox(height: 10),
        Text(
          _message,
          style: TextStyle(fontSize: 18, color: Colors.white70),
        ),
      ],
    ),
  ),
);
}
}

```

```

class EventDetailsScreen extends StatelessWidget {
  final String eventName;

```

```

  EventDetailsScreen({required this.eventName});

```

```

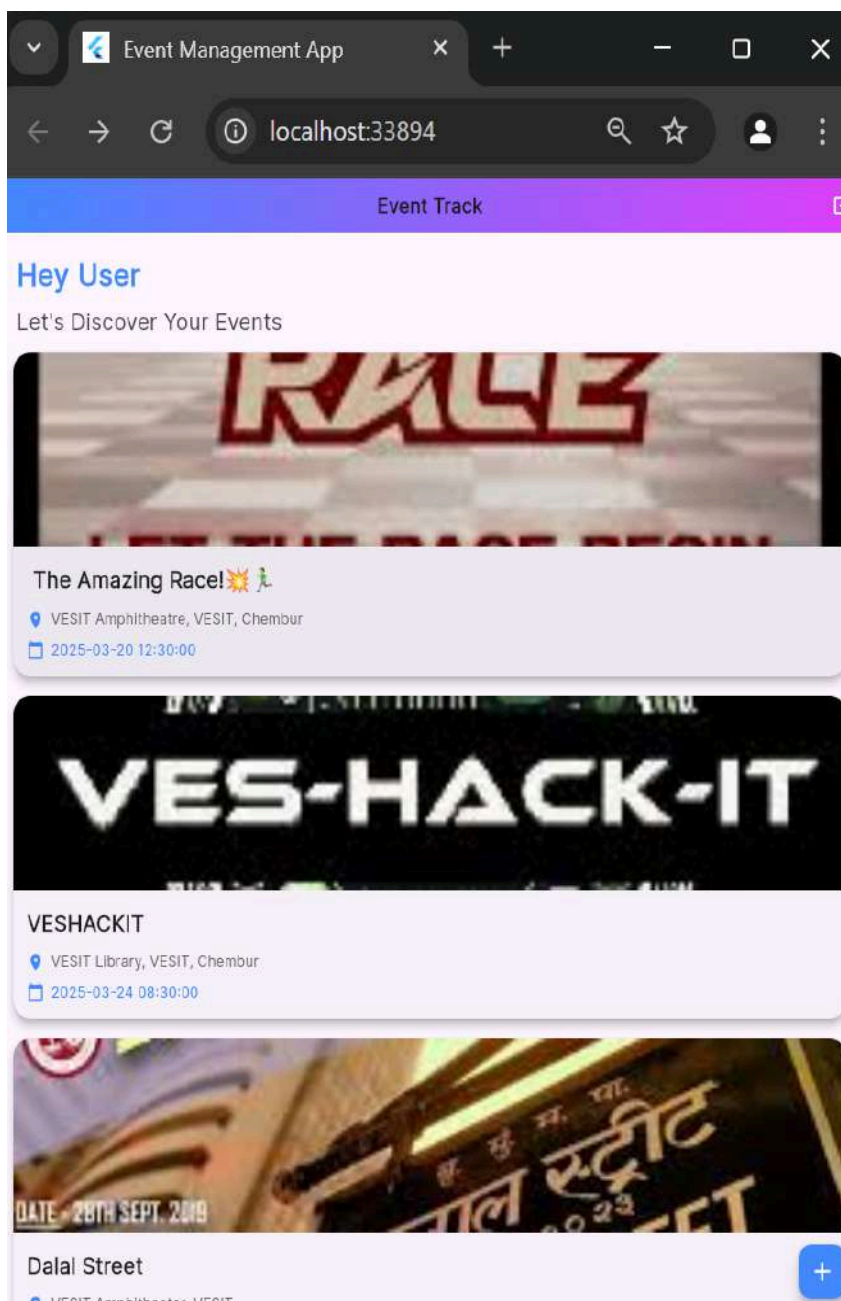
  @override

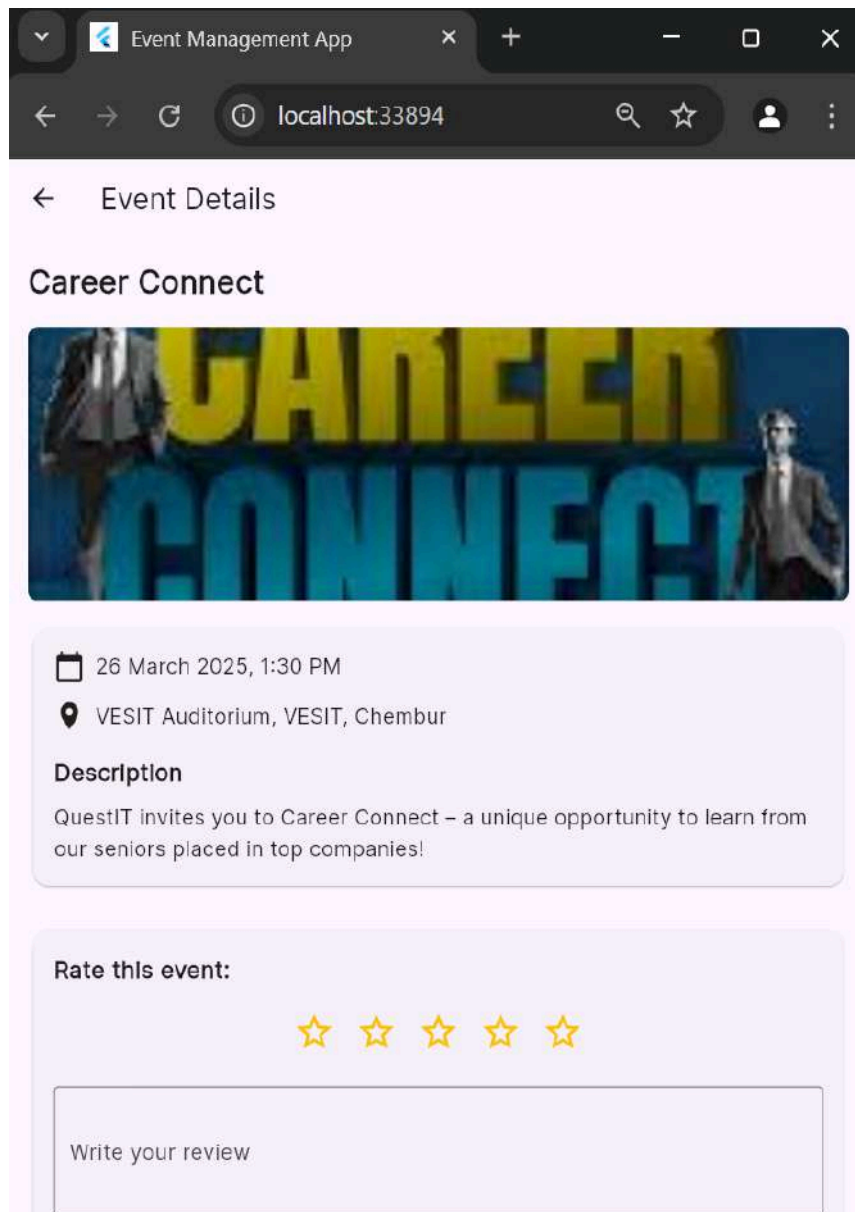
```



```
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: Text('Event Details')),
    body: Center(
      child: Text(
        'Details for $eventName',
        style: TextStyle(fontSize: 20, fontWeight: FontWeight.bold),
      ),
    ),
  ),
);
}
```

OUTPUT





Conclusion

Navigation allows movement between screens using Navigator. Routing helps structure navigation better using named routes. Gesture detection enables interactivity with user input. These features make Flutter apps more user-friendly.

Name: Sujal Tekwani

Class: D15B

Roll No: 59

Setting Up Firebase with Flutter for iOS and Android Apps

This document provides a detailed step-by-step guide on how to integrate Firebase with a Flutter project for both iOS and Android platforms. Firebase offers a suite of tools for app development, including analytics, authentication, cloud storage, and more. By following this guide, you will be able to set up Firebase in your Flutter app and start using its features.

Prerequisites

Before starting, ensure you have the following:

Flutter SDK installed on your machine.

Android Studio or Xcode for Android and iOS development, respectively.

A Firebase account (create one at firebase.google.com).

A Flutter project created (`flutter create project_name`).

Step 1: Create a Firebase Project

Go to the Firebase Console.

Click Add Project.

Enter a project name and follow the prompts to create the project.

Once the project is created, you will be redirected to the Firebase project dashboard.

Step 2: Add Firebase to Your Flutter Project

For Android

In the Firebase Console, click the Android icon to add an Android app to your Firebase project.

Enter your app's details:

Android package name: Find this in your `android/app/build.gradle` file under `applicationId`.

App nickname (optional): Add a nickname for your app.

Debug signing certificate SHA-1 (optional): If you need Firebase Authentication or Dynamic Links, add your SHA-1 key.

Click Register App.

Download the google-services.json file and place it in the android/app directory of your Flutter project.

Add the following dependencies to your android/build.gradle file:

```
buildscript {  
    dependencies {  
        classpath 'com.google.gms:google-services:4.3.15' // Use the latest version  
    }  
}
```

Add the following to the bottom of your android/app/build.gradle file:

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

For iOS

In the Firebase Console, click the iOS icon to add an iOS app to your Firebase project.

Enter your app's details:

iOS bundle ID: Find this in your Xcode project under Bundle Identifier.

App nickname (optional): Add a nickname for your app.

Click Register App.

Download the GoogleService-Info.plist file.

Open your Flutter project in Xcode.

Drag and drop the GoogleService-Info.plist file into the Runner directory in Xcode.

Ensure the file is added to the Runner target.

Add the following to your ios/Podfile:

```
platform :ios, '11.0' # or higher
```

Run pod install in the ios directory to install Firebase dependencies.

Step 3: Add Firebase Dependencies to Flutter

Open your pubspec.yaml file in your Flutter project.

Add the following dependencies under dependencies:

dependencies:

flutter:

 sdk: flutter

 firebase_core: latest_version # Required for Firebase integration

 firebase_analytics: latest_version # Optional: For analytics

 firebase_auth: latest_version # Optional: For authentication

 cloud_firestore: latest_version # Optional: For Firestore database

 firebase_storage: latest_version # Optional: For cloud storage

Run flutter pub get to install the dependencies.

Step 4: Initialize Firebase in Your Flutter App

Open your lib/main.dart file.

Import the Firebase Core package:

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

Initialize Firebase in the main function:

dart

Copy

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

Step 5: Test Firebase Integration

Run your app on an Android or iOS emulator/device.

Check the Firebase Console to ensure your app is connected and sending data (e.g., analytics events).

Step 6: Use Firebase Services

Now that Firebase is set up, you can start using its services in your Flutter app. For example:

Firebase Authentication: Add user authentication using email/password, Google Sign-In, etc.

Firebase Firestore: Store and retrieve data from a NoSQL database.

Firebase Storage: Upload and download files.

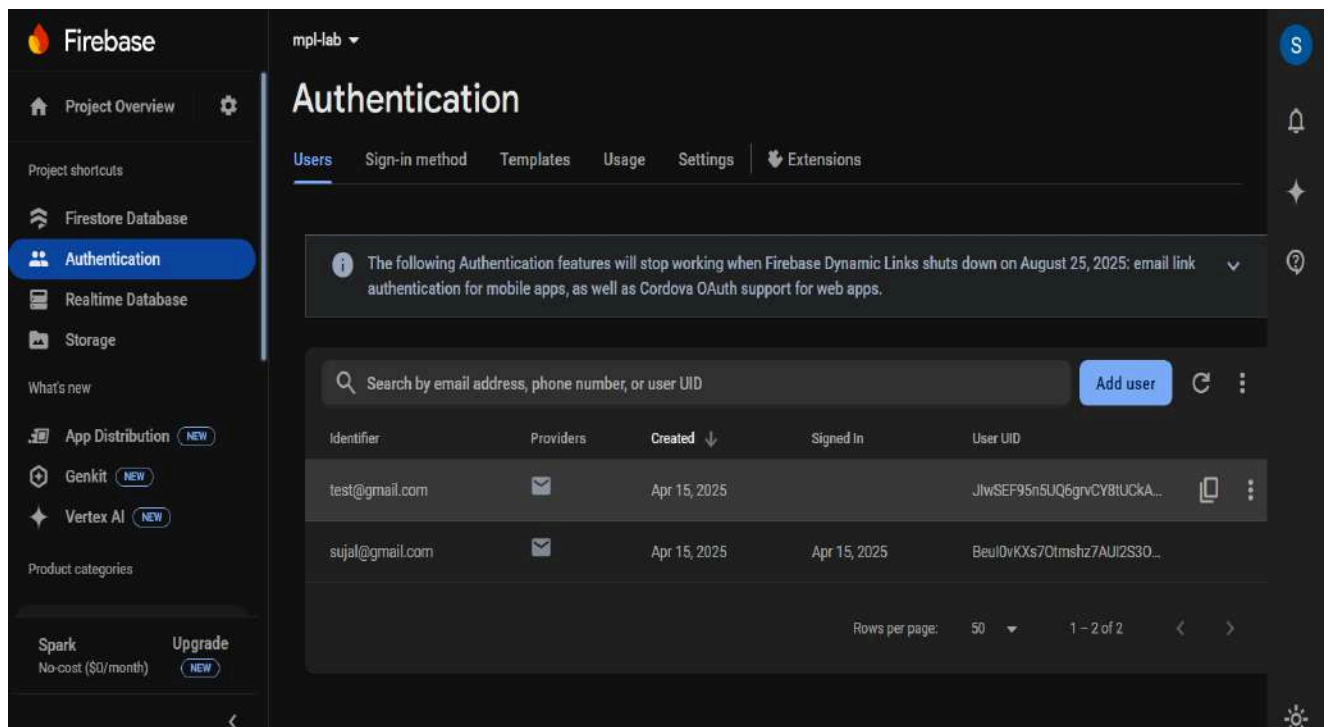
Firebase Analytics: Track user behavior and app usage.

Troubleshooting

Android: If you encounter issues with the google-services.json file, ensure it is placed in the correct directory (android/app).

iOS: If the app crashes on launch, ensure the GoogleService-Info.plist file is added to the Xcode project and the Runner target.

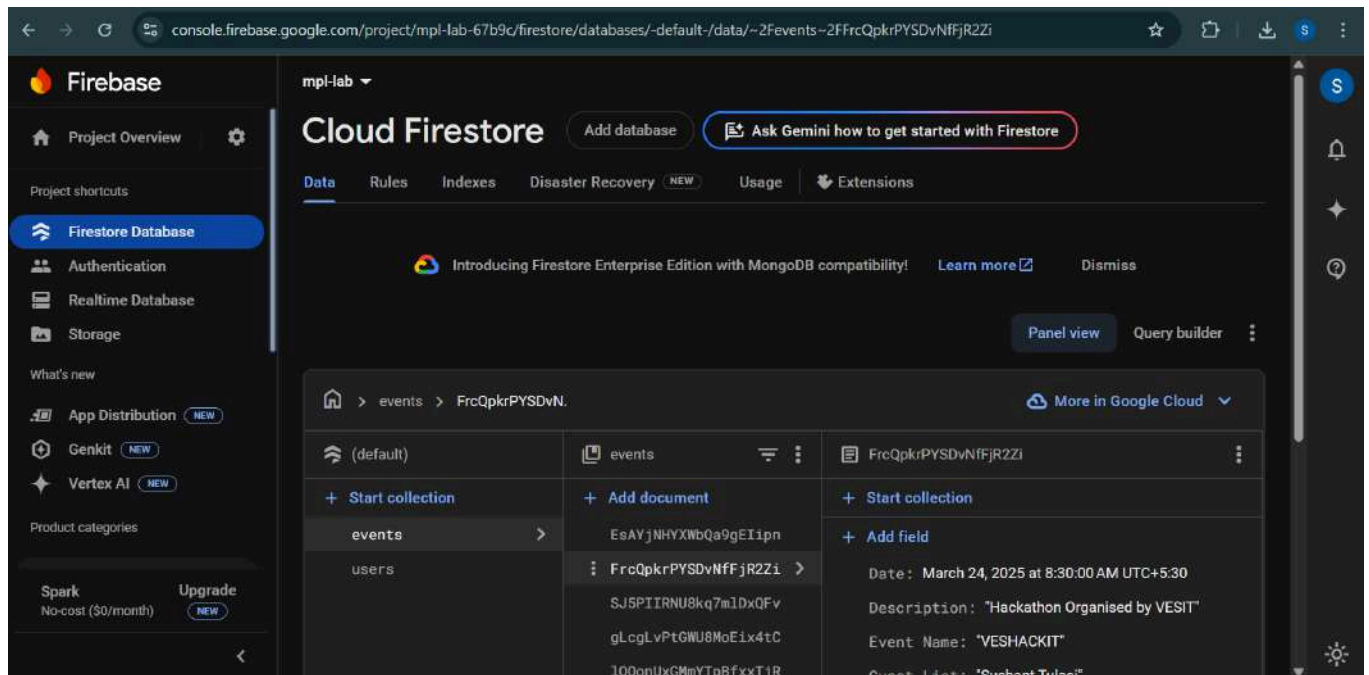
Dependencies: Always use the latest versions of Firebase plugins and ensure there are no version conflicts.



The screenshot displays the Firebase Authentication console for a project named 'mpl-lab'. The left sidebar contains navigation links for Project Overview, Project shortcuts (Firestore Database, Authentication, Realtime Database, Storage), What's new (App Distribution, Genkit, Vertex AI), and Product categories (Spark, Upgrade). The main content area is titled 'Authentication' and includes tabs for Users, Sign-in method, Templates, Usage, Settings, and Extensions. A warning message states: 'The following Authentication features will stop working when Firebase Dynamic Links shuts down on August 25, 2025: email link authentication for mobile apps, as well as Cordova OAuth support for web apps.' Below this is a search bar and an 'Add user' button. A table lists two users:

Identifier	Providers	Created ↓	Signed In	User UID
test@gmail.com		Apr 15, 2025		JlwSEF95n5UQ6gncY8tUCkA...
sujal@gmail.com		Apr 15, 2025	Apr 15, 2025	Beul0vKXs70tmshz7AUl2S30...

At the bottom right, it shows 'Rows per page: 50' and '1 - 2 of 2'.



Conclusion

You have successfully set up Firebase in your Flutter app for both iOS and Android platforms. You can now leverage Firebase's powerful features to enhance your app's functionality. For more details, refer to the official Firebase Flutter documentation. Replace latest_version with the actual version numbers of the Firebase plugins you are using. You can find the latest versions on pub.dev.

Name: Sujal.S.Tekwani

Class: D15B

Roll No:59

EXPERIMENT No. 7

Aim: To write meta data of your Ecommerce PWA in a Web app manifest file to enable add to homescreen feature

Theory:

Making a Vite React App a Progressive Web App (PWA)

A Progressive Web App (PWA) is a type of web application that provides a reliable, fast, and engaging user experience similar to a native mobile app. It leverages modern web technologies, including service workers, caching strategies, and a web app manifest, to enable offline functionality, push notifications, and an installable experience.

In the context of a web application, converting the app into a PWA involves integrating these key components:

1. Service Workers:

- Service workers act as a proxy between the browser and the network, enabling background processes such as caching and offline capabilities.
- They allow the app to load even when there is no internet connection by storing static assets locally.

2. Web App Manifest:

- The `manifest.json` file provides metadata about the app, including its name, icons, theme color, and display mode.
- This file helps the browser recognize the app as installable and controls its behavior when launched.

3. HTTPS Hosting

- A secure environment is required for service workers and manifest to work.
- Always host your PWA over **HTTPS** to ensure integrity and security.

Key Features of a PWA:

- **Offline Availability:** Allows users to access the app without an internet connection.
- **Fast Performance:** Uses caching strategies to reduce loading time.
- **App-Like Interface:** Can be installed on devices and launched in a standalone mode.
- **Secure & Reliable:** Served over HTTPS to prevent data tampering.
- **Background Sync & Push Notifications:** Keeps users engaged with real-time updates.

Steps to Make Your Web Application a PWA

Create a Web App Manifest File

- Go to the root or `public` folder of your project.
 - Create a new file named `manifest.json`.
 - Add metadata such as the app name, icons, theme color, and display mode.
 - This file helps browsers recognize your app as installable and enables the Add to Home Screen feature.
-

Link the Manifest in HTML

- Open your `index.html` file.
 - Add a `<link>` tag to reference the `manifest.json` file inside the `<head>` section.
 - Also, include a `<meta>` tag for the theme color.
 - This allows the browser to detect and use the manifest.
-

Register a Service Worker

- Create a new file named `service-worker.js` in your project root.
 - Add code to cache static assets and handle fetch events.
 - Service workers enable offline functionality and caching strategies.
-

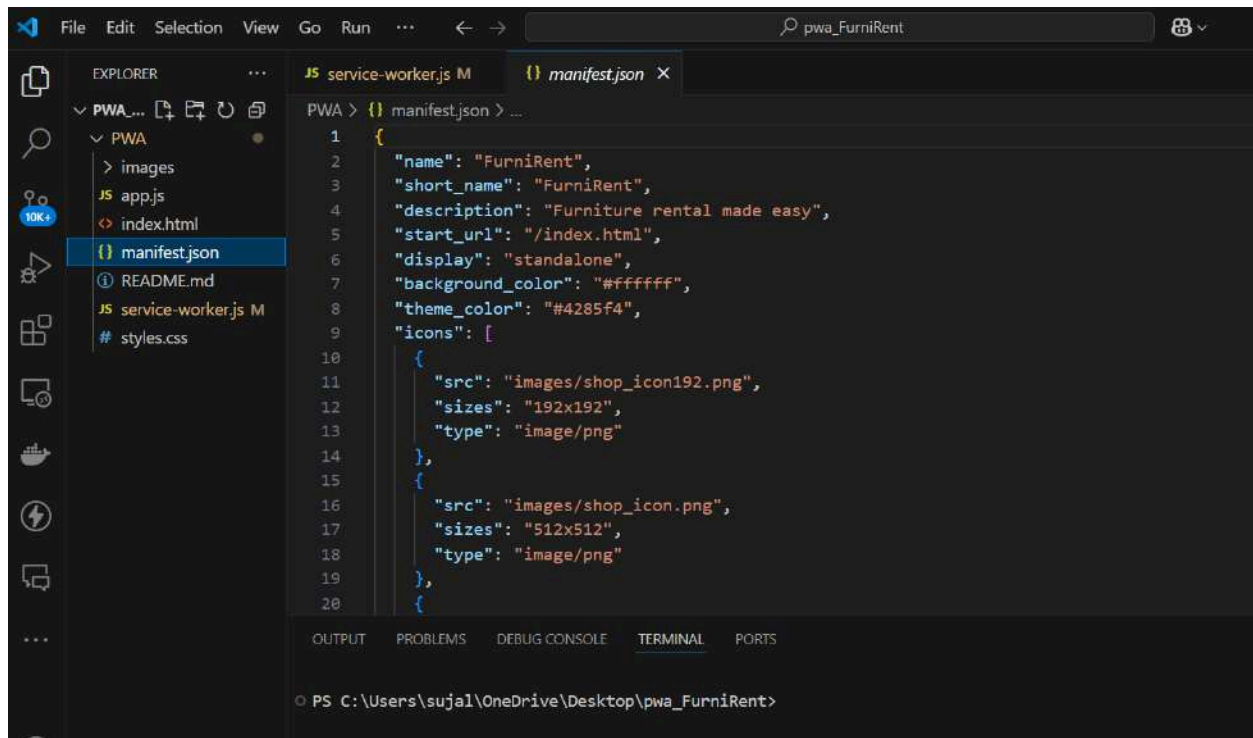
Add Service Worker Registration Script

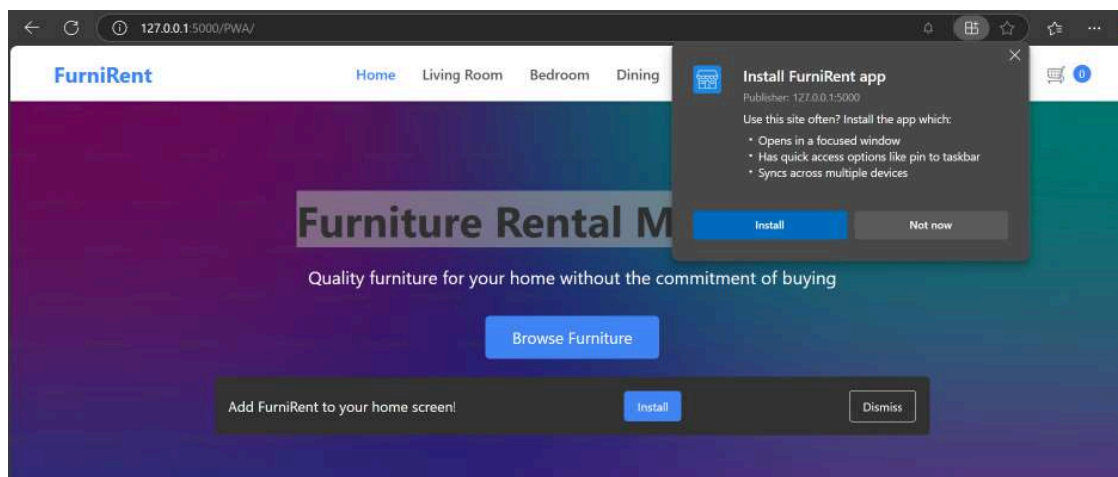
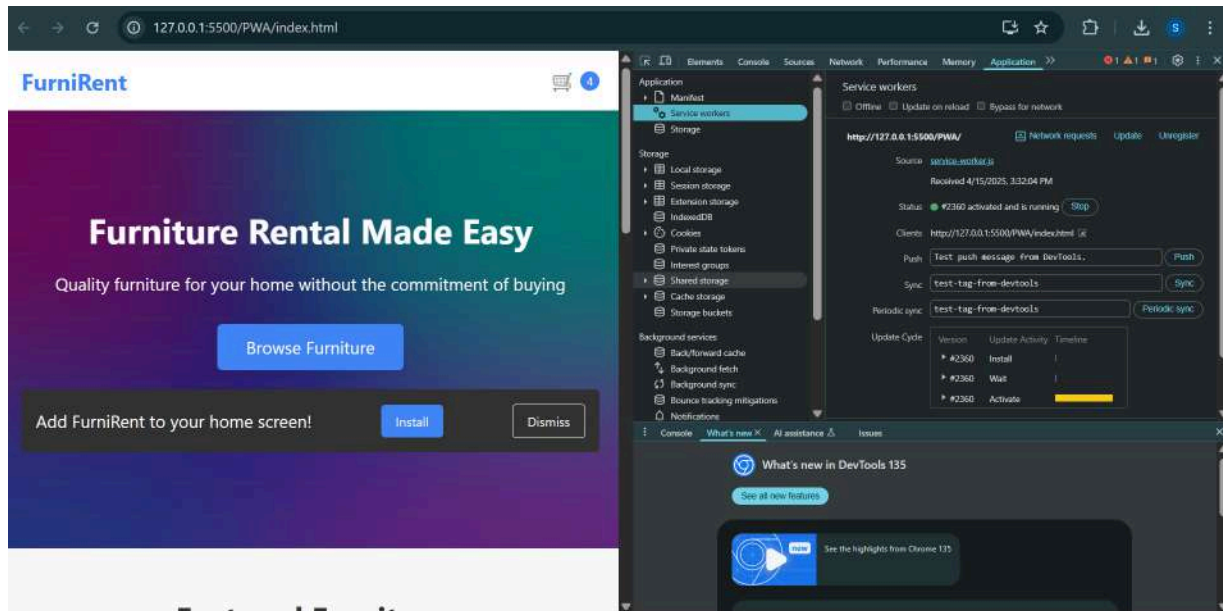
- Open your main JavaScript file (e.g., `main.js` or inside a `<script>` tag in `index.html`).
 - Add the logic to register the service worker when the page loads.
 - This ensures the service worker gets activated and starts handling caching.
-

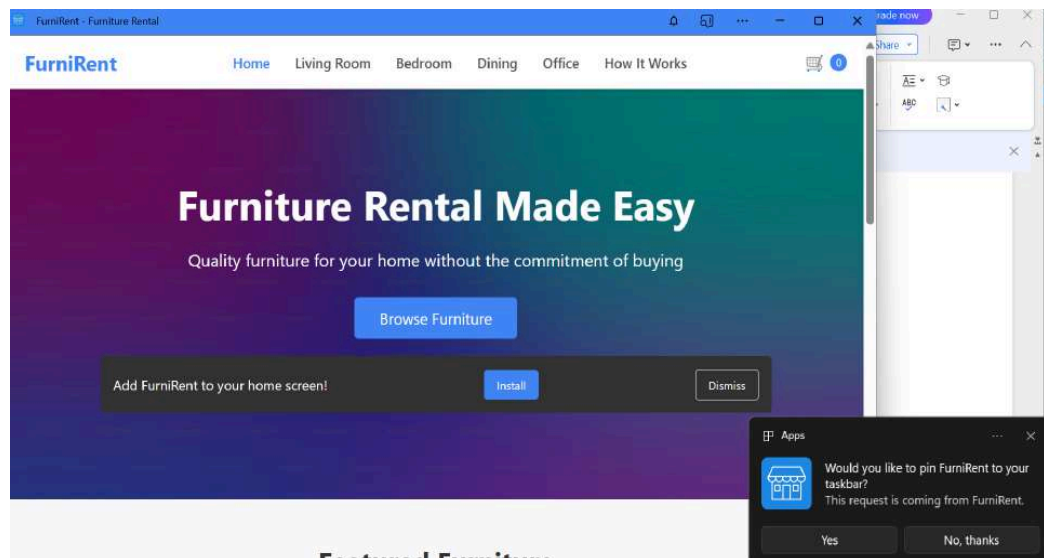
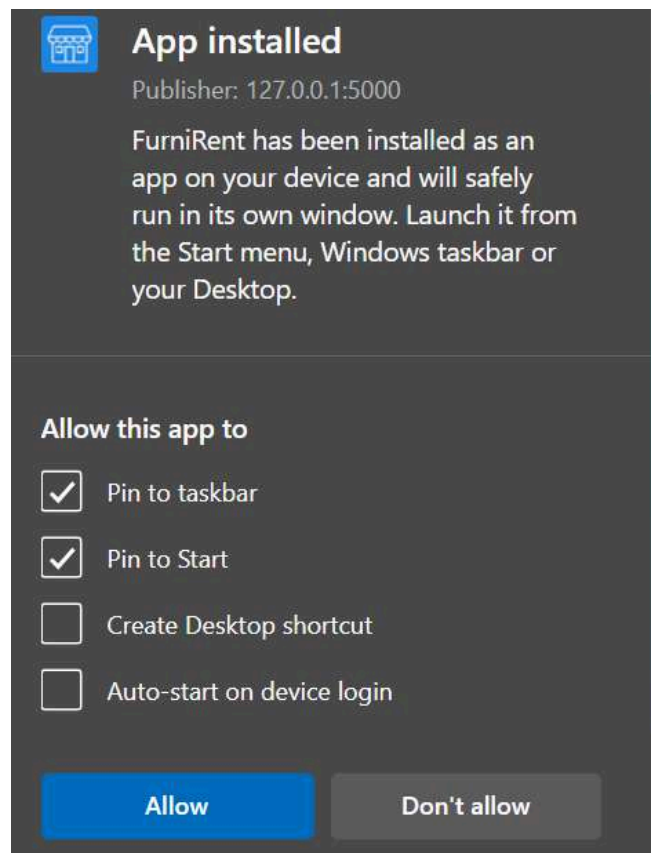
Serve Over HTTPS

- Host your web app using HTTPS to ensure security.
- Platforms like Netlify, Vercel, or GitHub Pages provide free HTTPS support.
- HTTPS is required for service workers and installability features.

OUTPUT :







Conclusion:

By following these steps, we successfully converted our Vite React application into a Progressive Web App. This enables offline accessibility, caching for better performance, and an installable app experience on mobile and desktop devices. Implementing PWA features enhances user engagement and makes the app function seamlessly even in limited network conditions.

Name: Sujal.S.Tekwani

Class: D15B

Roll No: 59

EXPERIMENT No. 8

Aim: To code and register a service worker, and complete the install and activation process for a new service worker for the E-commerce PWA.

Theory:

Service Worker

Service Worker is a script that works on browser background without user interaction independently. Also, It resembles a proxy that works on the user side. With this script, you can track network traffic of the page, manage push notifications and develop "offline first" web applications with Cache API.

Things to note about Service Worker:

- A service worker is a programmable network proxy that lets you control how network requests from your page are handled.
- Service workers only run over HTTPS. Because service workers can intercept network requests and modify responses, "man-in-the-middle" attacks could be very bad.
- The service worker becomes idle when not in use and restarts when it's next needed. You cannot rely on a global state persisting between events. If there is information that you need to persist and reuse across restarts, you can use IndexedDB databases.

What can we do with Service Workers?

- You can dominate Network Traffic

You can manage all network traffic of the page and do any manipulations. For example, when the page requests a CSS file, you can send plain text as a response or when the page requests an HTML file, you can send a png file as a response. You can also send a true response too.

- You can Cache

You can cache any request/response pair with Service Worker and Cache API and you can access these offline content anytime.

- You can manage Push Notifications

You can manage push notifications with Service Worker and show any information message to the user.

- You can Continue Although Internet connection is broken, you can start any process with Background Sync of Service Worker.

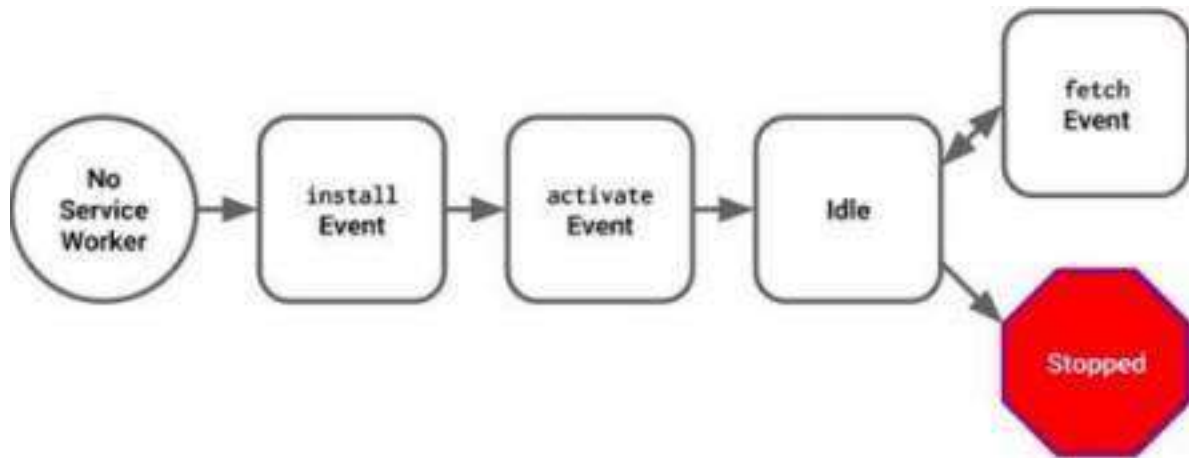
What can't we do with Service Workers?

- You can't access the Window

You can't access the window, therefore, You can't manipulate DOM elements. But, you can communicate to the window through post Message and manage processes that you want.

- You can't work it on 80 Port

Service Worker just can work on HTTPS protocol. But you can work on localhost during development. Service Worker Cycle



A

service worker goes through three steps in its life cycle:

- Registration
- Installation
- Activation

Registration

Installation

Activation

To install a service worker, you need to register it in your main JavaScript code. Registration tells the browser where your service worker is located, and to start installing it in the background. Let's look at an example:

main.js

```

if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/service-worker.js')
    .then(function(registration) {
      console.log('Registration successful, scope is:', registration.scope);
    })
    .catch(function(error) {
      console.log('Service worker registration failed, error:', error);
    });
}
  
```

This code starts by checking for browser support by examining `navigator.serviceWorker`. The service worker is then registered with `navigator.serviceWorker.register`, which returns a promise that resolves when the service worker has been successfully registered. The scope of the service worker is then logged with `registration.scope`. If the service worker is already installed, `navigator.serviceWorker.register` returns the registration object of the currently active service worker.

The scope of the service worker determines which files the service worker controls, in other words, from which path the service worker will intercept requests. The default scope is the location of the service worker file, and extends to all directories below. So if `service-worker.js` is located in the root directory, the service worker will control requests from all files at this domain. You can also set an arbitrary scope by passing in an additional parameter when registering. For example:

main.js

```

navigator.serviceWorker.register('/service-worker.js', {
  scope: '/app/'
})
  
```

```
});
```

In this case we are setting the scope of the service worker to `/app/`, which means the service worker will control requests from pages like `/app/`, `/app/lower/` and `/app/lower/lower`, but not from pages like `/app` or `/`, which are higher.

If you want the service worker to control higher pages e.g. `/app` (without the trailing slash) you can indeed change the scope option, but you'll also need to set the `Service-Worker-Allowed` HTTP Header in your server config for the request serving the service worker script.

main.js

```
navigator.serviceWorker.register('/app/service-worker.js', {  
  scope: '/app'  
});
```

Installation

Once the browser registers a service worker, installation can be attempted. This occurs if the service worker is considered to be new by the browser, either because the site currently doesn't have a registered service worker, or because there is a byte difference between the new service worker and the previously installed one.

A service worker installation triggers an `install` event in the installing service worker. We can include an `install` event listener in the service worker to perform some task when the service worker installs. For instance, during the `install`, service workers can precache parts of a web app so that it loads instantly the next time a user opens it (see caching the application shell). So, after that first load, you're going to benefit from instant repeat loads and your time to interactivity is going to be even better in those cases. An example of an installation event listener looks like this:

service-worker.js

```
// Listen for install event, set callback  
self.addEventListener('install', function(event) {  
  // Perform some task  
});
```

Activation

Once a service worker has successfully installed, it transitions into the activation stage. If there are any open pages controlled by the previous service worker, the new service worker enters a waiting state. The new service worker only activates when there are no longer any pages loaded that are still using the old service worker. This ensures that only one version of the service worker is running at any given time.

When the new service worker activates, an `activate` event is triggered in the activating service worker. This event listener is a good place to clean up outdated caches (see the [Offline Cookbook](#) for an example).

service-worker.js

```
self.addEventListener('activate', function(event) {  
  // Perform some task  
});
```

Once activated, the service worker controls all pages that load within its scope, and starts listening for events from those pages. However, pages in your app that were loaded before the service worker activation will not be under service worker control. The new service worker will

only take over when you close and reopen your app, or if the service worker calls `clients.claim()`. Until then, requests from this page will not be intercepted by the new service worker. This is intentional as a way to ensure consistency in your site.

CODE :

sw.js

```
/**
 * Copyright 2018 Google Inc. All Rights Reserved.
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *   http://www.apache.org/licenses/LICENSE-2.0
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

// If the loader is already loaded, just stop.
if (!self.define) {
  let registry = {};

  // Used for `eval` and `importScripts` where we can't get script URL by other means.
  // In both cases, it's safe to use a global var because those functions are synchronous.
  let nextDefineUri;

  const singleRequire = (uri, parentUri) => {
    uri = new URL(uri + ".js", parentUri).href;
    return registry[uri] || (

      new Promise(resolve => {
        if ("document" in self) {
          const script = document.createElement("script");
          script.src = uri;
          script.onload = resolve;
          document.head.appendChild(script);
        } else {
          nextDefineUri = uri;
          importScripts(uri);
          resolve();
        }
      })
    )
  }
}
```

```

    .then(() => {
      let promise = registry[uri];
      if (!promise) {
        throw new Error(`Module ${uri} didn't register its module`);
      }
      return promise;
    })
  );
};

self.define = (depsNames, factory) => {
  const uri = nextDefineUri || ("document" in self ? document.currentScript.src : "") ||
location.href;
  if (registry[uri]) {
    // Module is already loading or loaded.
    return;
  }
  let exports = {};
  const require = depUri => singleRequire(depUri, uri);
  const specialDeps = {
    module: { uri },
    exports,
    require
  };
  registry[uri] = Promise.all(depsNames.map(
    depName => specialDeps[depName] || require(depName)
  )).then(deps => {
    factory(...deps);
    return exports;
  });
};
}

define(['./workbox-d9a5ed57'], (function (workbox) { 'use strict';

self.skipWaiting();
workbox.clientsClaim();

/**
 * The precacheAndRoute() method efficiently caches and responds to
 * requests for URLs in the manifest.
 * See https://goo.gl/S9QRab
 */
workbox.precacheAndRoute([
  {url: "registerSW.js",

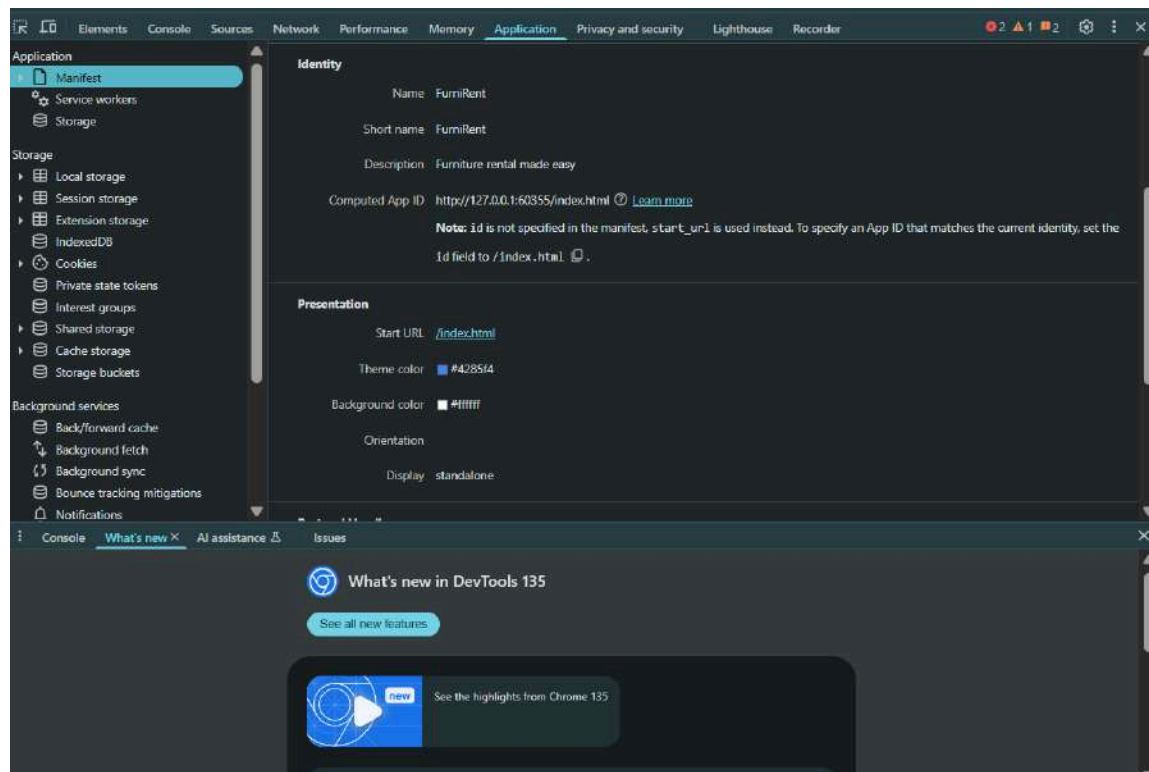
```

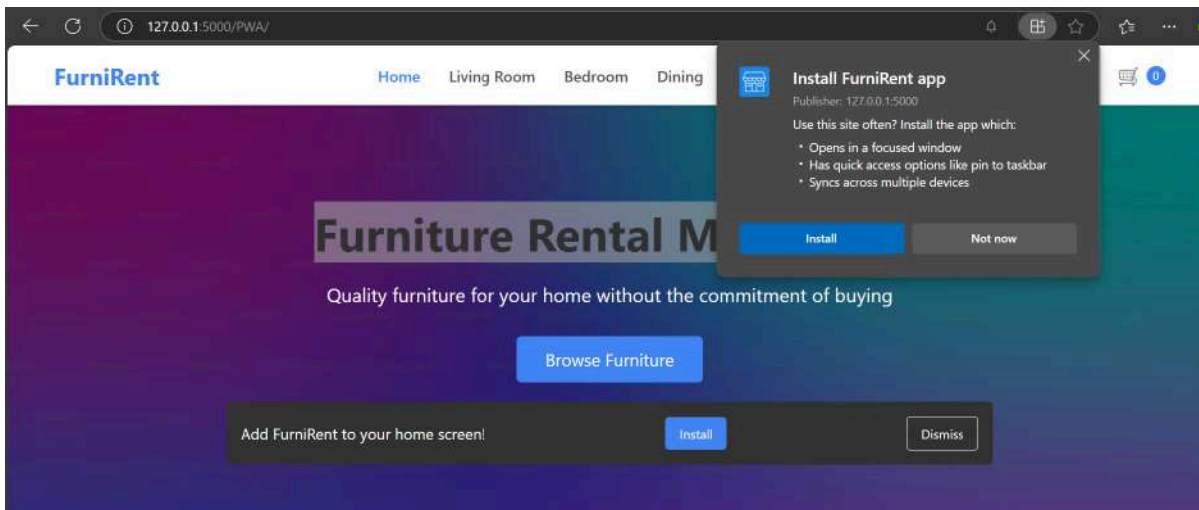
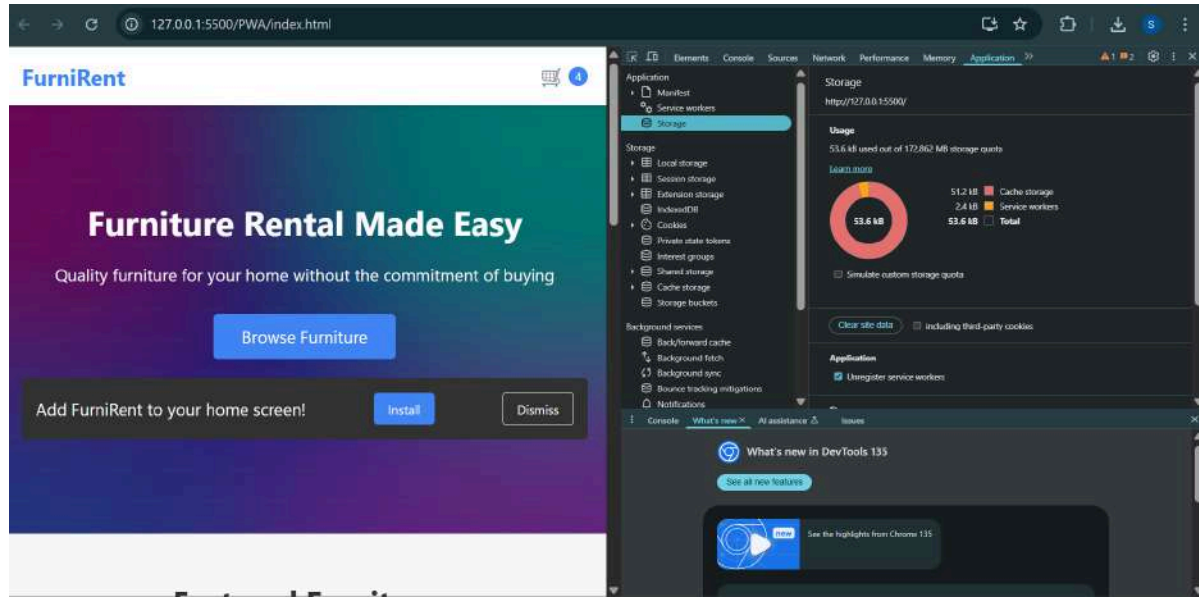
```

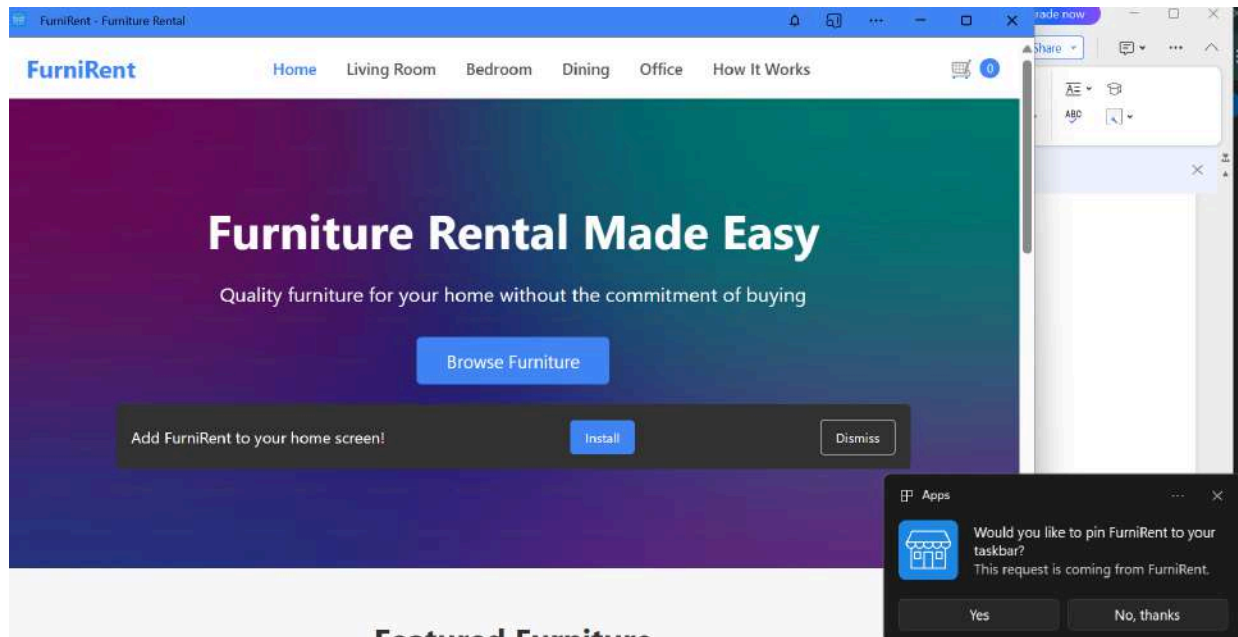
    "revision": "3ca0b8505b4bec776b69afdba2768812"
  }, {
    "url": "index.html",
    "revision": "0.7e7jl0981k"
  }, {});
workbox.cleanupOutdatedCaches();
workbox.registerRoute(new
workbox.NavigationRoute(workbox.createHandlerBoundToURL("index.html"), {
  allowlist: [/^V$/]
}));
workbox.registerRoute(/\.(?:png|jpg|jpeg|svg)$/ , new workbox.CacheFirst({
  "cacheName": "images-cache",
  plugins: [new workbox.ExpirationPlugin({
    maxEntries: 50,
    maxAgeSeconds: 2592000
  })]
}), 'GET');
});

```

OUTPUT :







CONCLUSION :

Service workers are essential for PWAs, enabling offline access, caching, and efficient network request handling. In this implementation, the service worker is registered, installed, and activated using **Workbox**, ensuring seamless updates and improved performance. By precaching assets and using runtime caching strategies, it enhances page load speed, reduces server load, and allows access to content even without an internet connection. This makes the e-commerce PWA more reliable, responsive, and user-friendly.

Name:Sujal.S.Tekwani

Class:D15B

Roll No:59

EXPERIMENT No. 9

Implementing Service Worker Events (Fetch, Sync, Push) for E-Commerce PWA

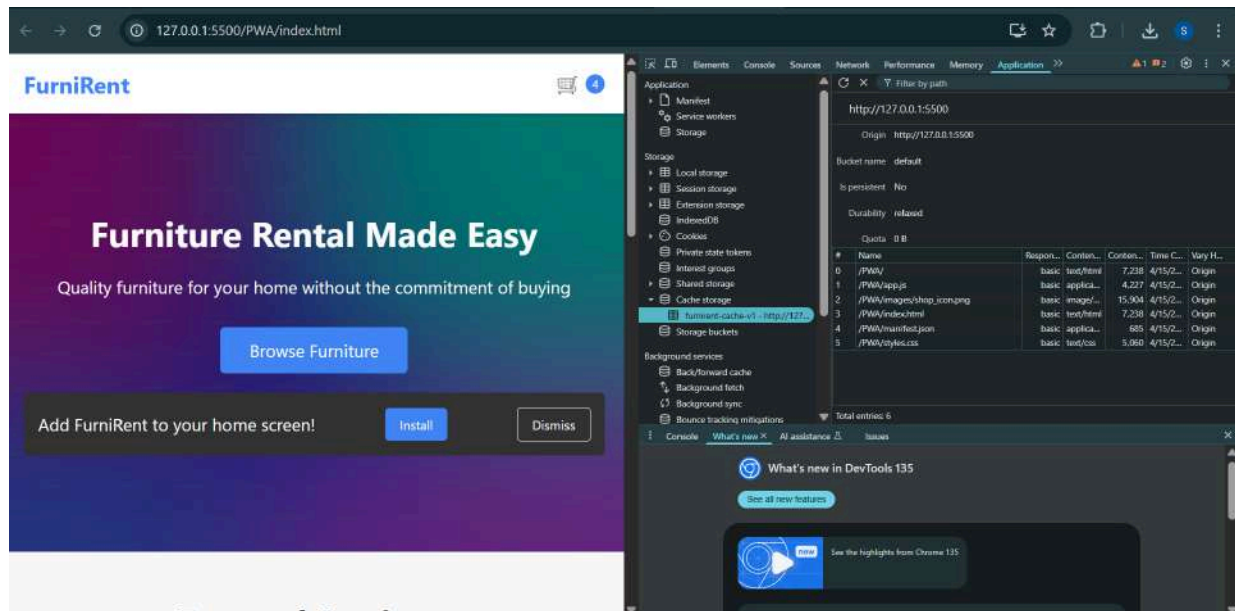
Progressive Web Apps (PWAs) are web applications that offer app-like experiences through modern web capabilities. One of the key components of a PWA is the service worker, which enables features like offline access, background sync, and push notifications. In this document, we will explore how to implement service worker events such as `fetch`, `sync`, and `push` in the context of an e-commerce application. Below is a sample implementation.

1. Caching Static Assets Using Install Event

The `install` event is triggered when the service worker is installed. During this phase, essential files are cached to enable offline access.

```
const CACHE_NAME = "campquest-v1";
const ASSETS_TO_CACHE = [
  "/",
  "/index.html",
  "/src/main.jsx",
  "/CampQuest.svg",
  "/manifest.json",
];

self.addEventListener("install", (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME).then((cache) => {
      return cache.addAll(ASSETS_TO_CACHE);
    })
  );
});
```



2. Handling Fetch Requests

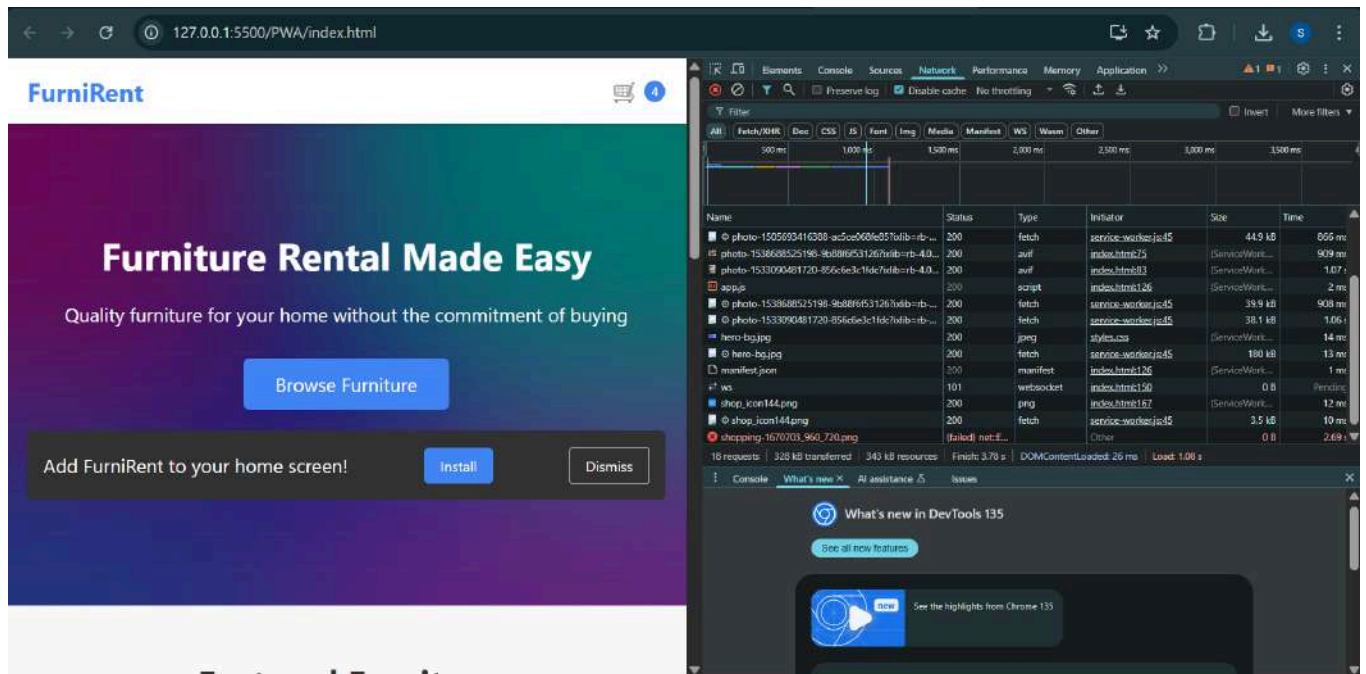
The `fetch` event intercepts network requests. We use this event to implement a cache-first or network-first strategy depending on the URL path.

```
self.addEventListener("fetch", (event) => {
  const url = new URL(event.request.url);
  if (url.pathname.startsWith("/campgrounds")) {
    event.respondWith(
      fetch(event.request)
        .then((response) => {
          const responseClone = response.clone();
          caches.open(CACHE_NAME).then((cache) => {
            cache.put(event.request, responseClone);
          });
          return response;
        })
        .catch(() => {
          return caches.match(event.request);
        })
    );
  }
});
```

```

} else {
  event.respondWith(
    caches.match(event.request).then((response) => {
      return response || fetch(event.request);
    })
  );
}
});

```



3. Background Sync (Conceptual Example)

The `sync` event is used to defer actions until the user has stable connectivity. For an e-commerce app, you could use this to sync cart data or orders.

```

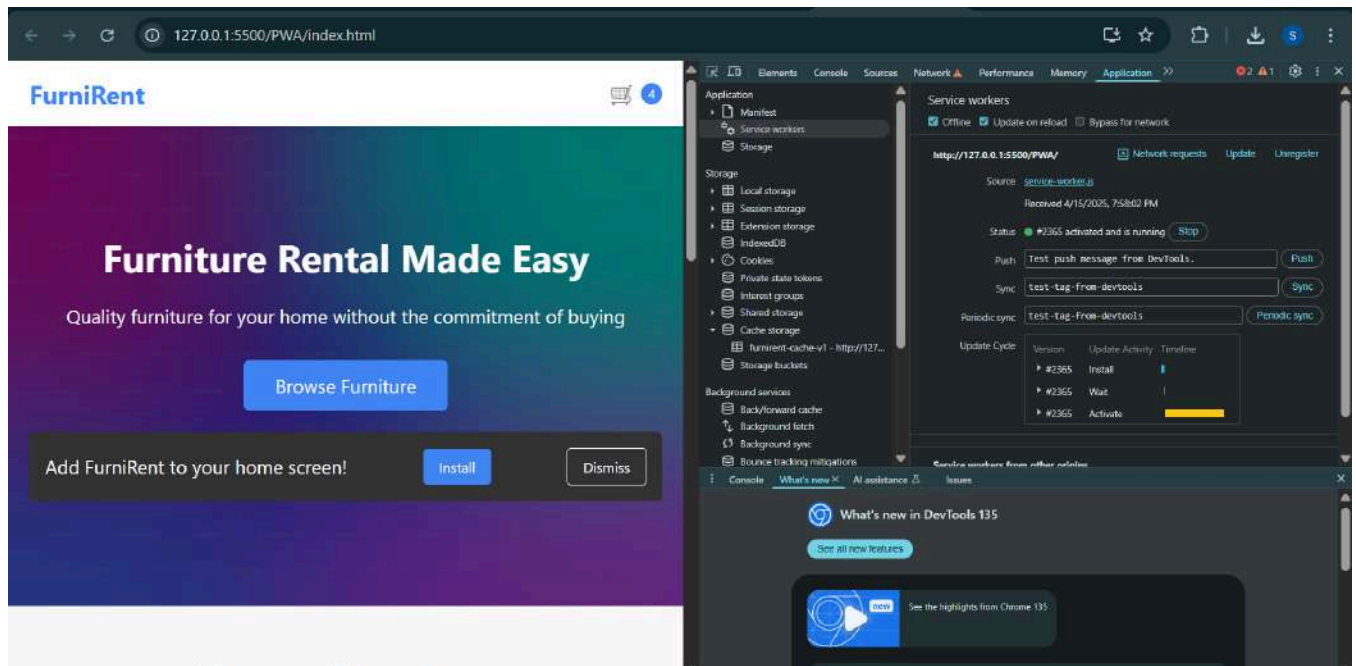
self.addEventListener("sync", (event) => {
  if (event.tag === "sync-cart") {
    event.waitUntil(
      // Logic to sync cart data with server
    );
  }
});

```


4. Push Notifications (Conceptual Example)

The `push` event is triggered when a push message is received. This could be used to notify users of new deals or order status updates.

```
self.addEventListener("push", (event) => {
  const data = event.data.json();
  const options = {
    body: data.body,
    icon: "/icon.png",
  };
  event.waitUntil(
    self.registration.showNotification(data.title, options)
  );
});
```



Conclusion

Service workers are powerful tools in building resilient and engaging e-commerce PWAs. By handling `install`, `fetch`, `sync`, and `push` events effectively, you can create a seamless experience for users, even in offline or low-connectivity scenarios.

Name: Sujal.S.Tekwani

Class: D15B

Roll: 59

EXPERIMENT No. 10

GitHub Pages Documentation

Introduction

GitHub Pages is a free web hosting service provided by GitHub that allows you to host static websites directly from a GitHub repository. It's perfect for portfolios, documentation, project pages, or any static content.

This guide will walk you through the process of setting up and publishing your website using GitHub Pages.

Prerequisites

- A GitHub account
 - A repository containing your static website (HTML, CSS, JS)
 - Basic knowledge of Git and GitHub
-

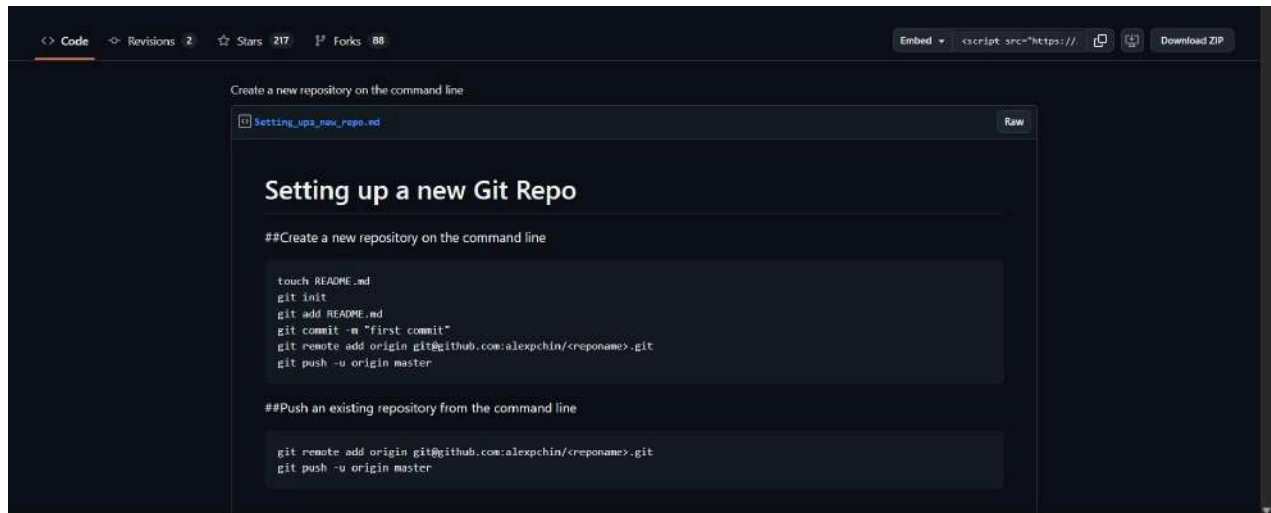
Steps to Deploy Your Website with GitHub Pages

Step 1: Create or Use an Existing Repository

If you don't have a repository yet, create one:

1. Go to github.com

2. Click on **New repository**
3. Name your repository (e.g., `my-portfolio`)
4. Initialize it with a README (optional)

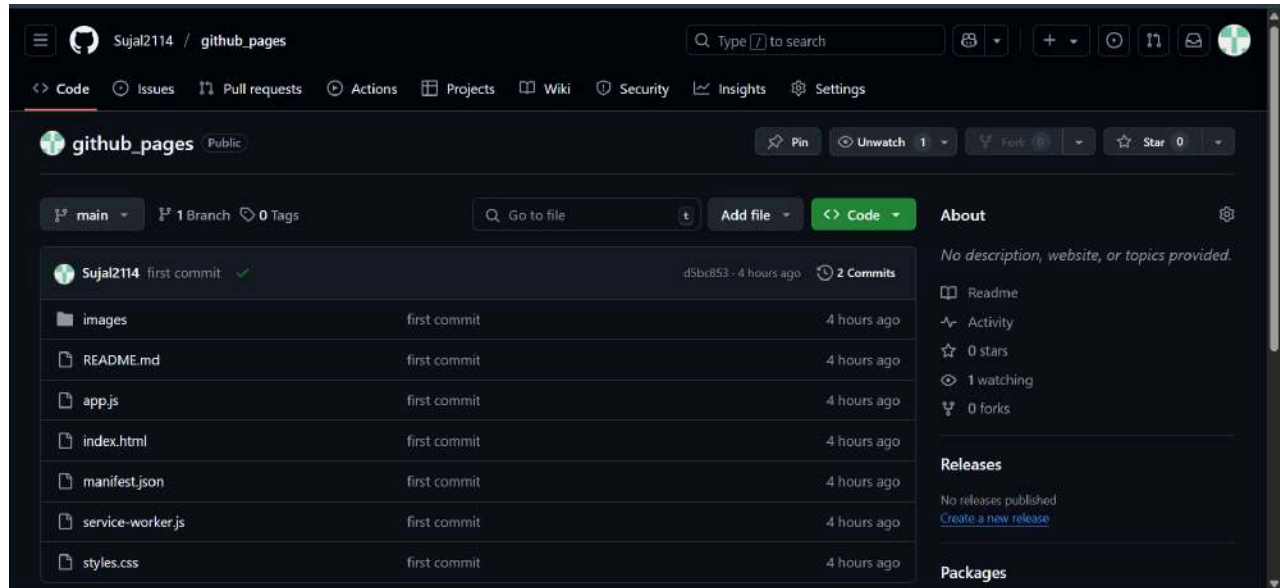


Step 2: Upload Your Website Files

Make sure your repository contains the files you want to publish, such as `index.html`.

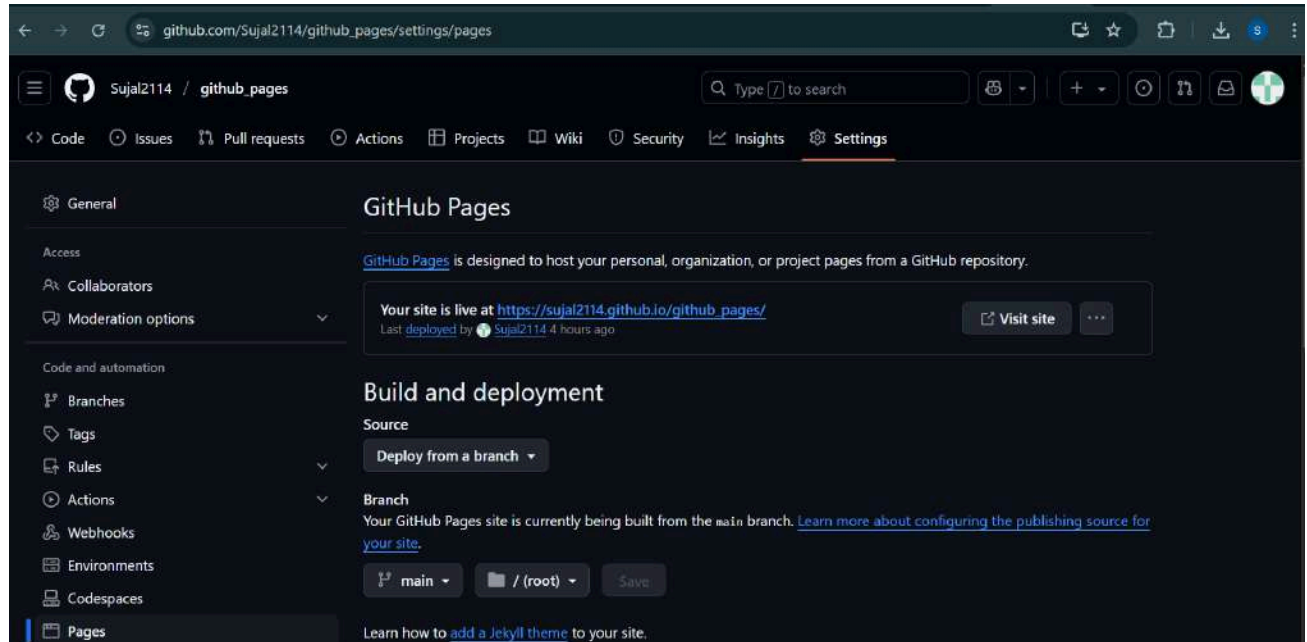
You can either:

- Drag and drop files on the web interface
- Use Git commands (`git add`, `git commit`, `git push`) from your local machine



Step 3: Enable GitHub Pages

1. Go to your repository
2. Click on the **Settings** tab
3. Scroll down to **Pages** in the left menu
4. Under **Source**, select the branch (usually `main`) and folder (e.g., `/root` or `/docs`)
5. Click **Save**

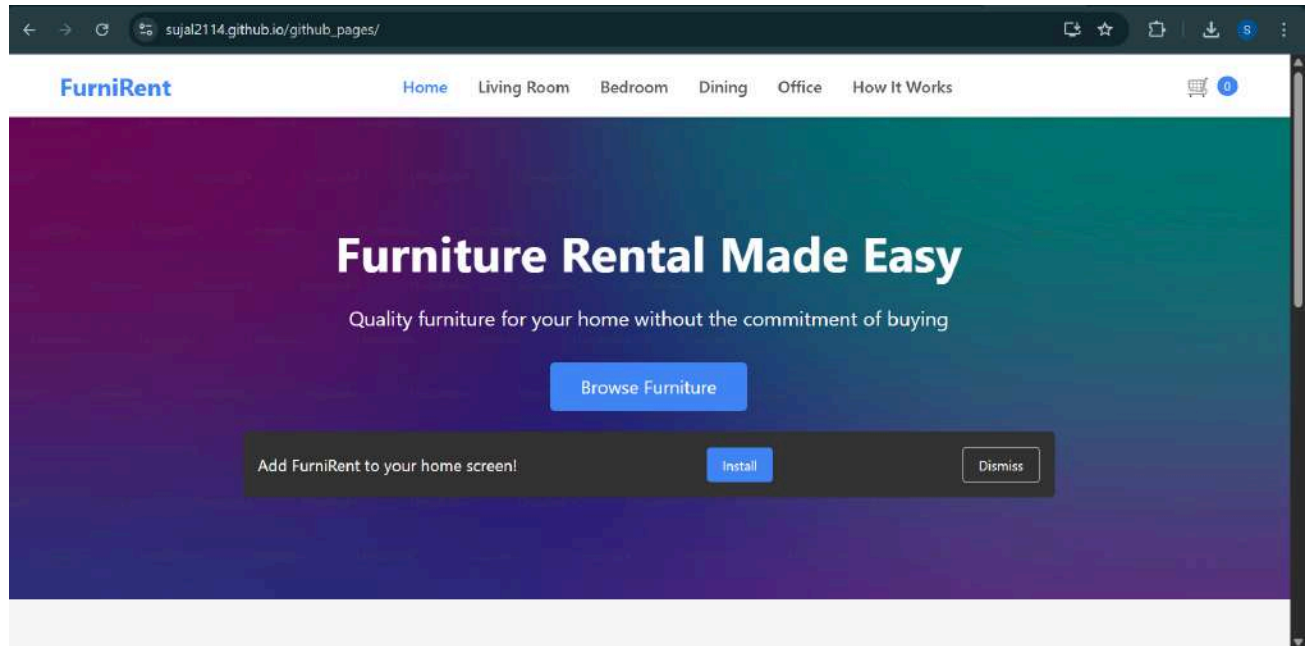


Step 4: Access Your Live Website

After enabling GitHub Pages:

- Wait a few seconds
- You'll see a link like:
https://sushant0412.github.io/github_pages/

Click it to view your live site 🎉



Tips

- Your homepage must be named `index.html`
- You can use a custom domain by configuring the **Custom domain** section under GitHub Pages settings
- Use relative paths for assets to avoid 404 errors

Conclusion

GitHub Pages provides an easy, fast, and free way to publish static websites directly from your GitHub repo. Whether you're showcasing a project or building a personal site, it's a powerful tool in your web development journey.

Name: Sujal.S.Tekwani

Class: D15B

Roll No:59

EXPERIMENT No. 11

Google Lighthouse Extension Documentation

What is Lighthouse?

Lighthouse is an open-source tool developed by Google to audit web pages for performance, accessibility, SEO, best practices, and Progressive Web App (PWA) compatibility. It helps developers improve the quality of their websites.

Lighthouse can be accessed:

- As a **Chrome DevTools tab** (built-in)
- As a **Chrome extension**
- From the command line (`lighthouse`)
- In **PageSpeed Insights**

Why Use Lighthouse?

Lighthouse is commonly used to:

- Evaluate page **performance** (loading speed, render times)
- Improve **accessibility** (for users with disabilities)
- Follow **SEO best practices**

- Detect **common coding issues**
 - Ensure PWA compliance (if applicable)
-

How to Use Lighthouse in Chrome Inspect Element

Step-by-step Guide

1. Open Your Website in Chrome

Go to the webpage you want to audit.



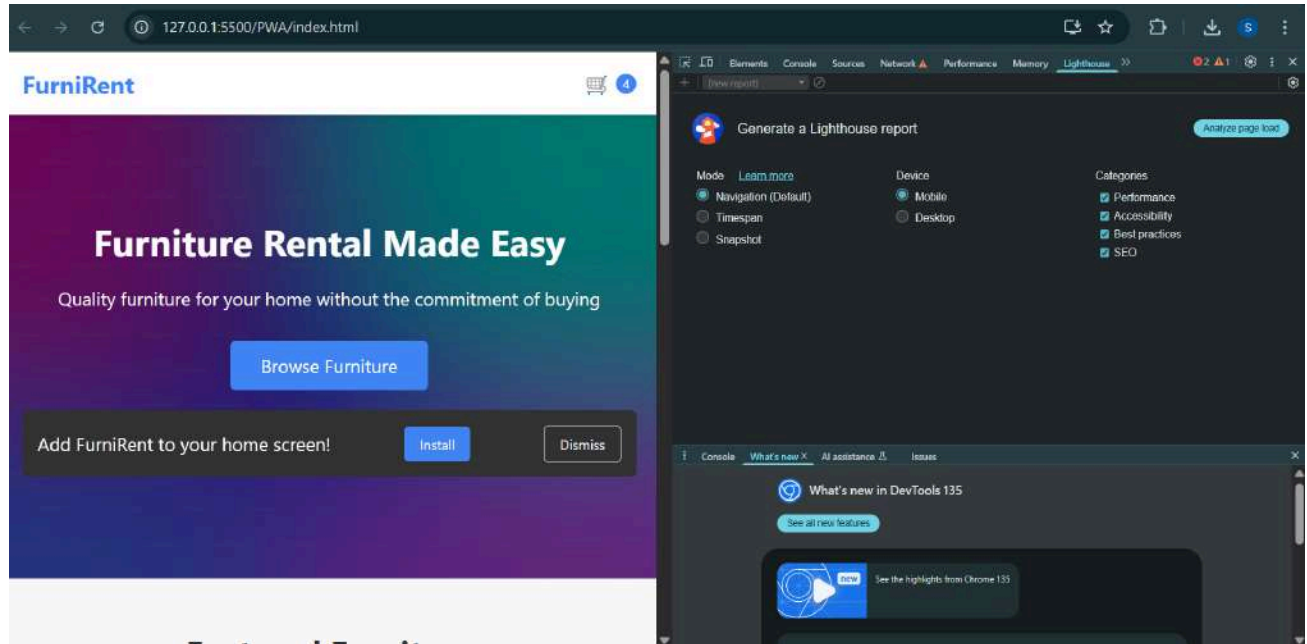
Insert Screenshot of your site opened in Chrome

2. Open Chrome DevTools

- Right-click anywhere on the page → click **Inspect**, OR
 - Press `Ctrl + Shift + I` (Windows) or `Cmd + Option + I` (Mac)
-

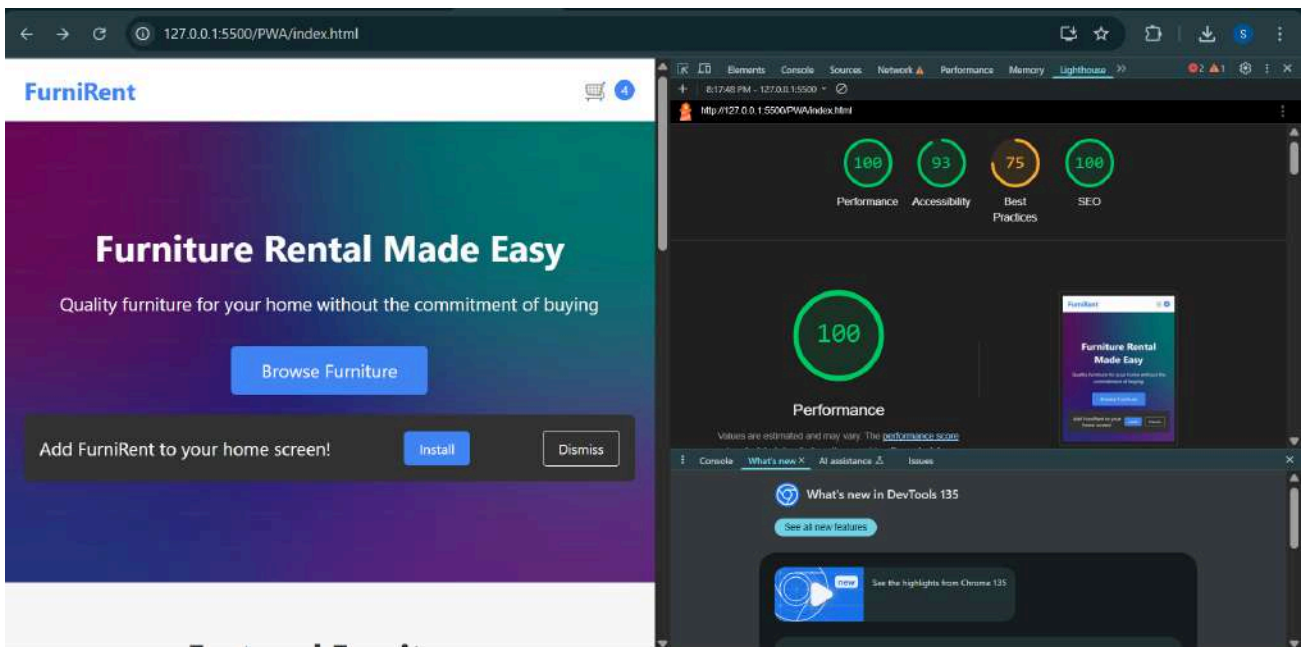
3. Configure Lighthouse Audit

- Choose device type: **Mobile** or **Desktop**
- Select categories: Performance, Accessibility, Best Practices, SEO, PWA
- Click **Analyze page load**



4. View the Report

Lighthouse will run its audit and generate a score report in a few seconds.



Understanding the Scores

Metric	Description
Performance	Measures speed, loading time, and responsiveness
Accessibility	Checks usability for assistive technologies (screen readers, etc.)
Best Practices	Analyzes common coding issues, HTTPS, errors
SEO	Reviews metadata, alt tags, and other ranking factors
PWA	Tests service workers, offline mode, installability (if PWA is present)

Conclusion

Lighthouse is an essential tool for modern web developers. It provides a comprehensive report on your website's strengths and weaknesses, helping you optimize user experience and site performance with actionable insights.

Start using it regularly during development to catch issues early and ship high-quality web apps

