Name: Varun Gupta

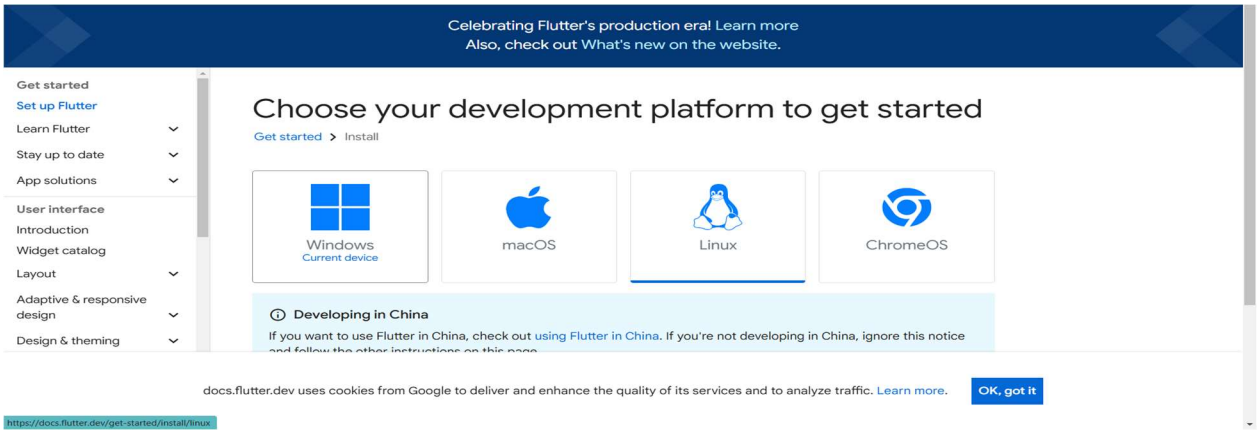
Class: D15B

Roll: 18

EXP 1: Installation and Configuration of Flutter Environment.

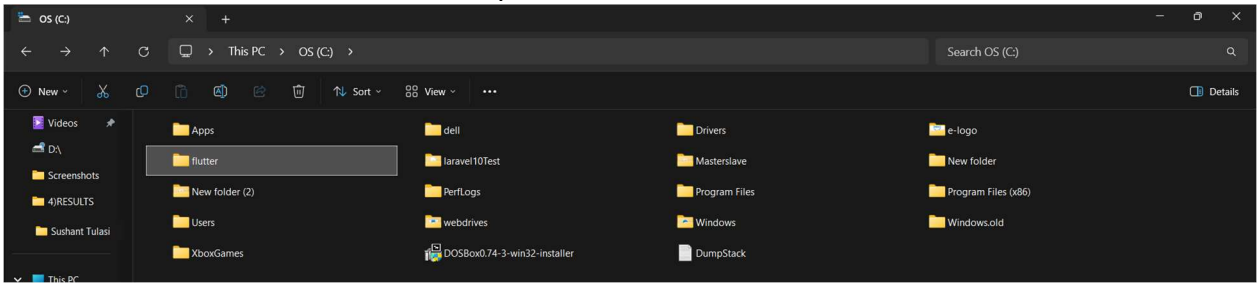
Install the Flutter SDK

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.



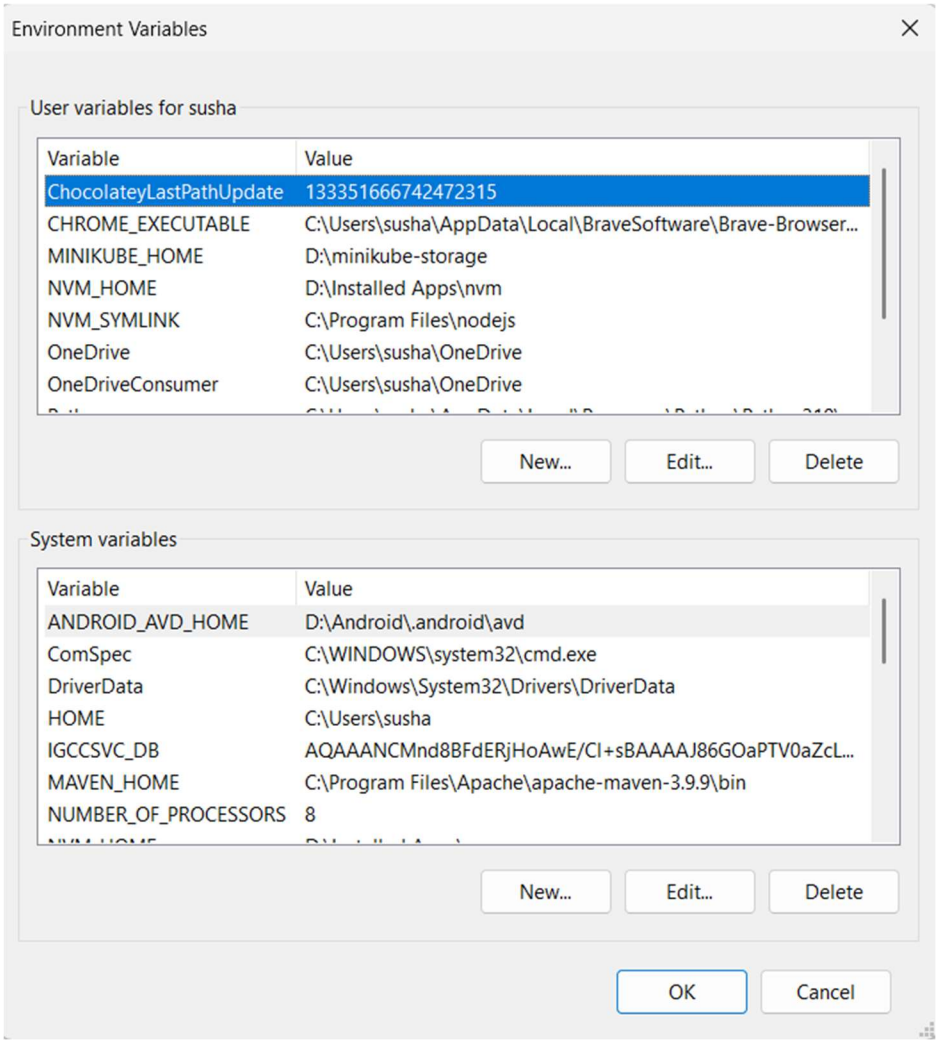
Step 2: Next, to download the latest Flutter SDK, click on the Windows icon. Here, you will find the download link for SDK.

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.

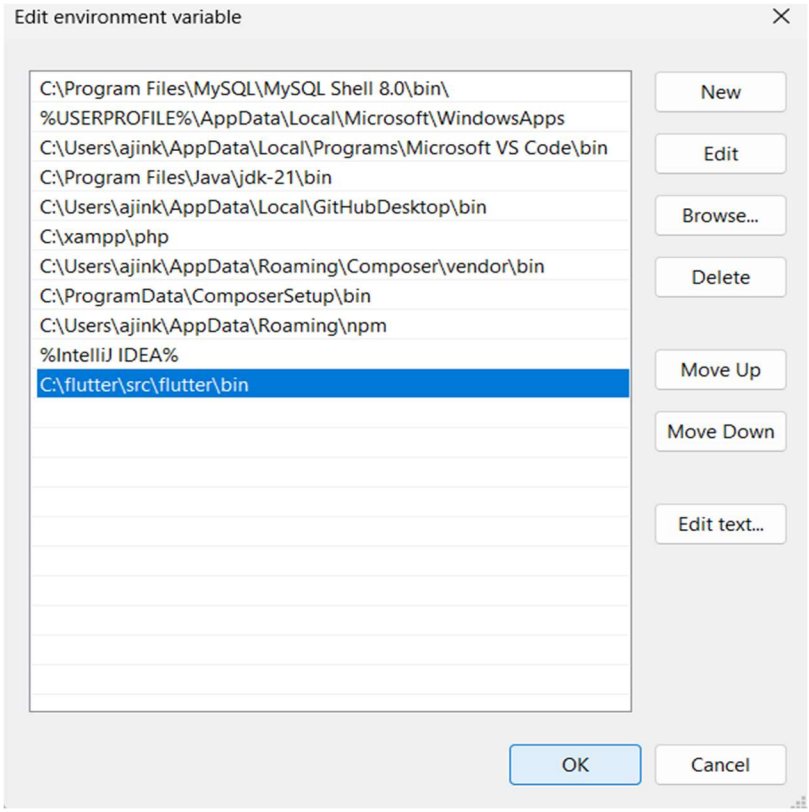


Step 4: To run the Flutter command in the regular windows console, you need to update the system path to include the flutter bin directory. The following steps are required to do this:

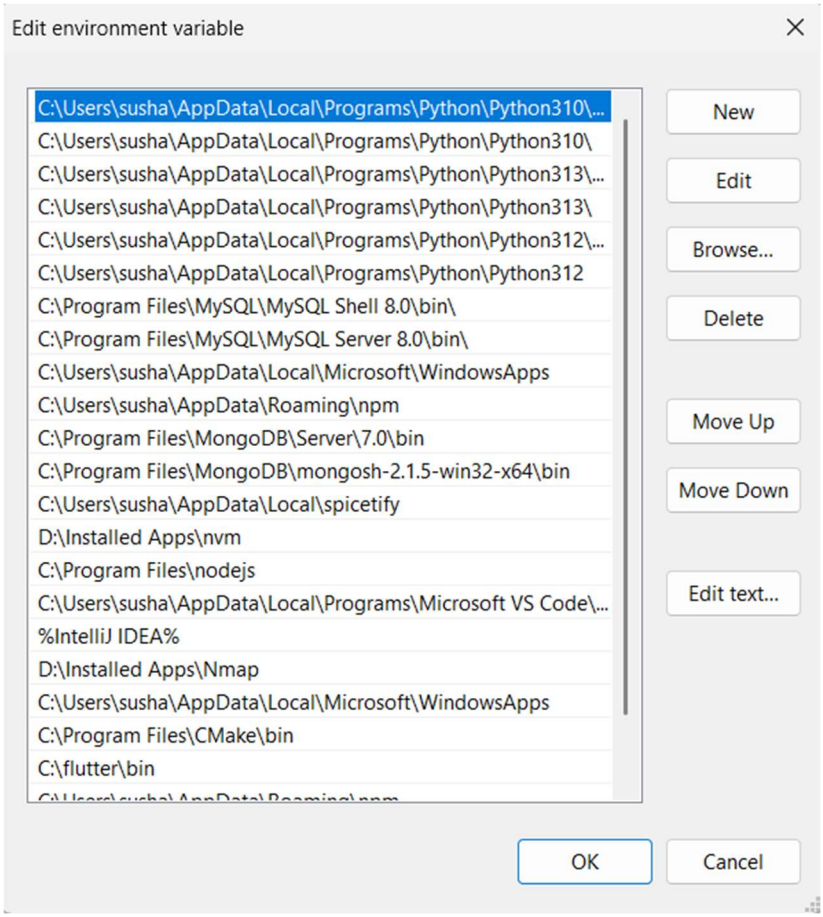
Step 4.1: Go to MyComputer properties -> advanced tab -> environment variables. You will get the following screen.



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.

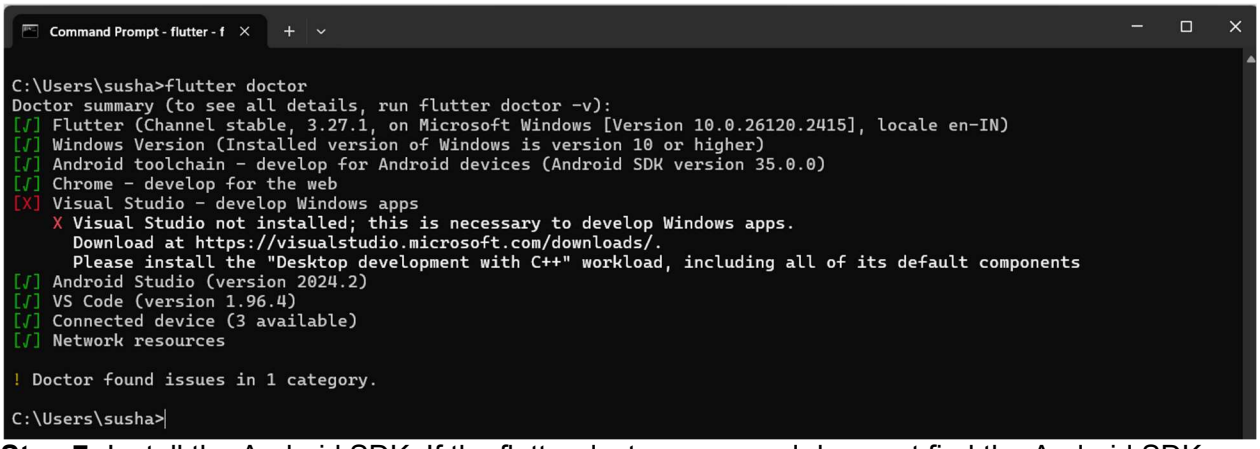


Step 5: Now, run the $ flutter command in command prompt.

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.

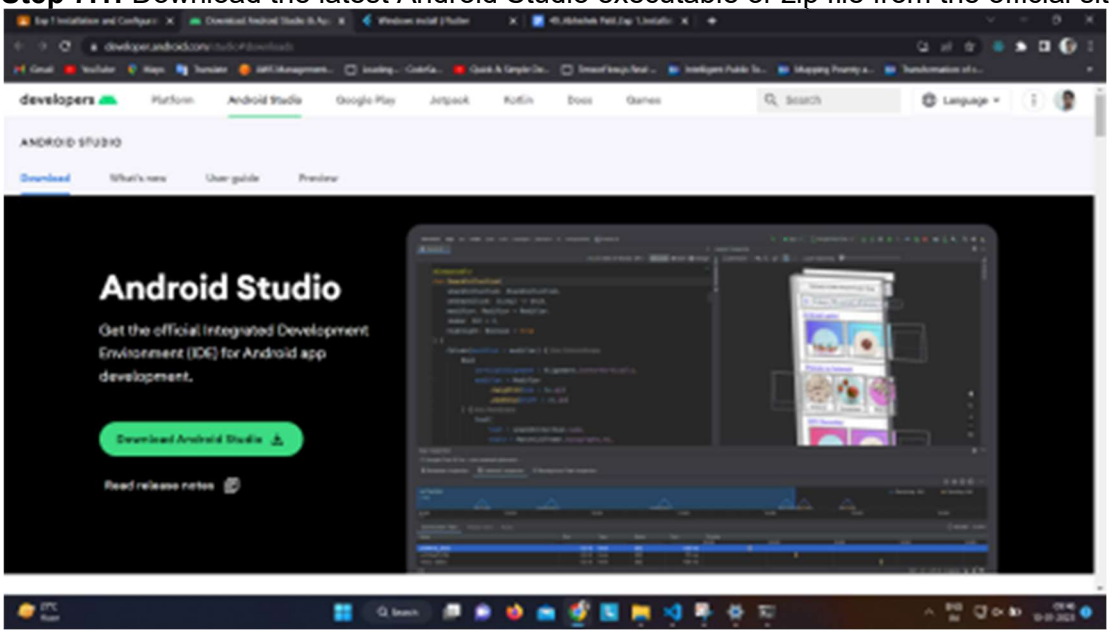
```
Microsoft Windows [Version 10.0.26120.2415]
(c) Microsoft Corporation. All rights reserved.

C:\Users\susha>flutter

  A new version of Flutter is available!

  To update to the latest version, run "flutter upgrade".

Manage your Flutter app development.

Common commands:

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

  flutter run [options]
    Run your Flutter application on an attached device or in an emulator.

Usage: flutter <command> [arguments]

Global options:
-h, --help                  Print this usage information.
-v, --verbose               Noisy logging, including all shell commands executed.
                            If used with "--help", shows hidden options. If used with "flutter doctor", shows additional
                            diagnostic information. (Use "-vv" to force verbose logging in those cases.)
-d, --device-id             Target device id or name (prefixes allowed).
    --version               Reports the version of this tool.
    --enable-analytics      Enable telemetry reporting each time a flutter or dart command runs.
```

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.

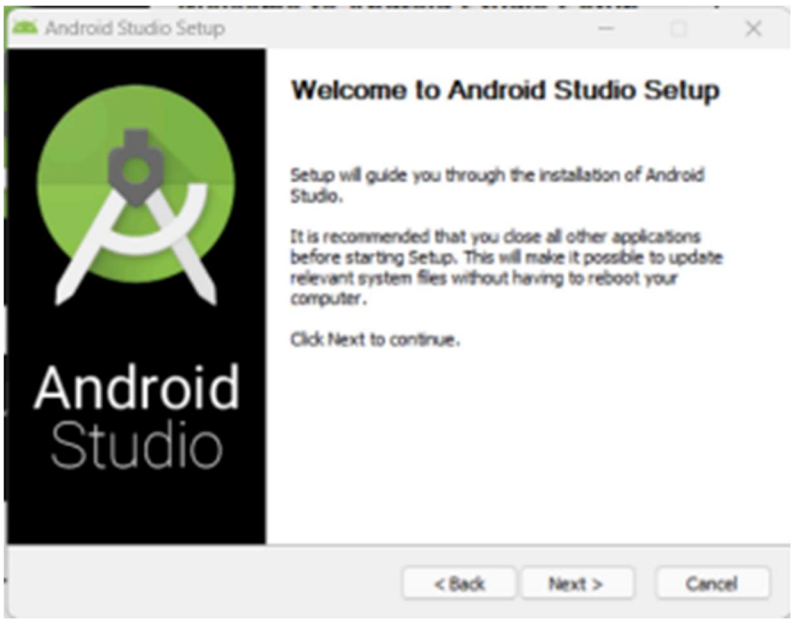


Step 7: Install the Android SDK. If the flutter doctor command does not find the Android SDK tool in your system, then you need first to install the Android Studio IDE. To install Android Studio IDE, do the following steps.

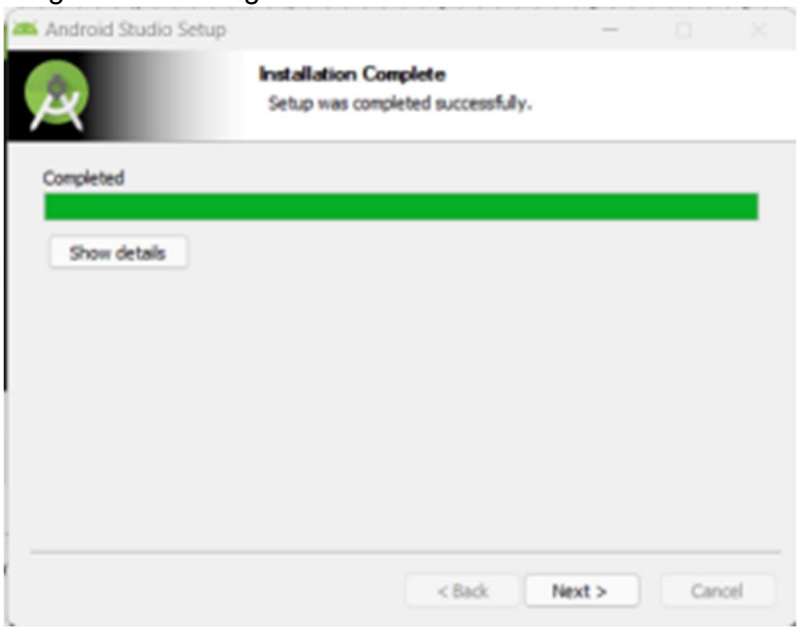
Step 7.1: Download the latest Android Studio executable or zip file from the official site.



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

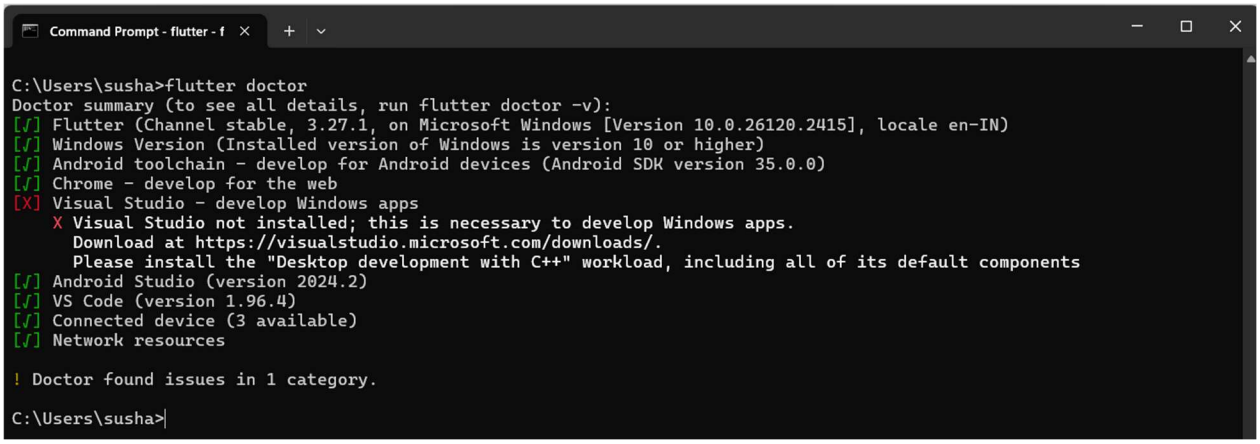


Step 7.3: Follow the steps of the installation wizard. Once the installation wizard completes, you will get the following screen.



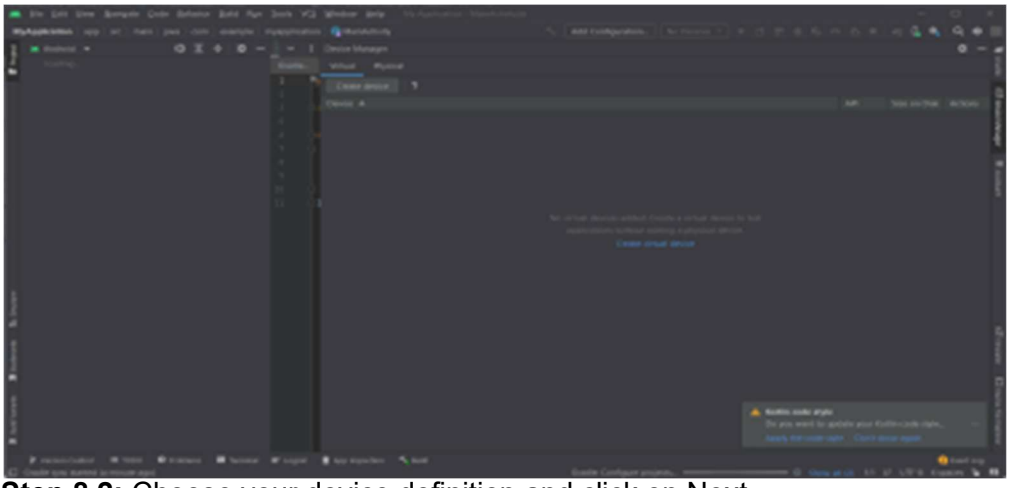
Step 7.4: In the above screen, click Next-> Finish. Once the Finish button is clicked, you need to choose the 'Don't import Settings option' and click OK. It will start the Android Studio.

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

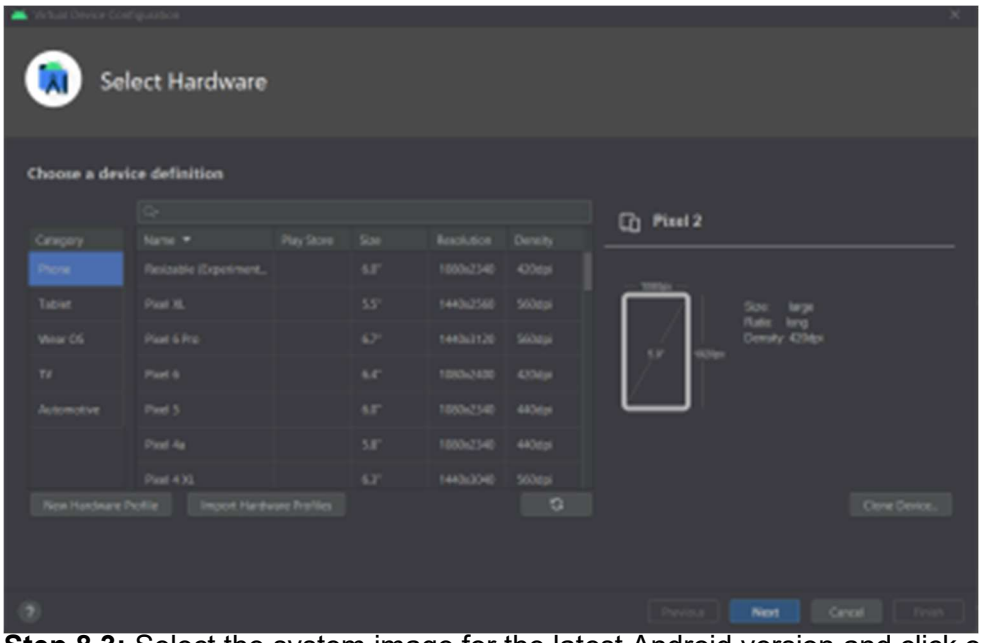


Step 8: Next, you need to set up an Android emulator. It is responsible for running and testing the Flutter application.

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.

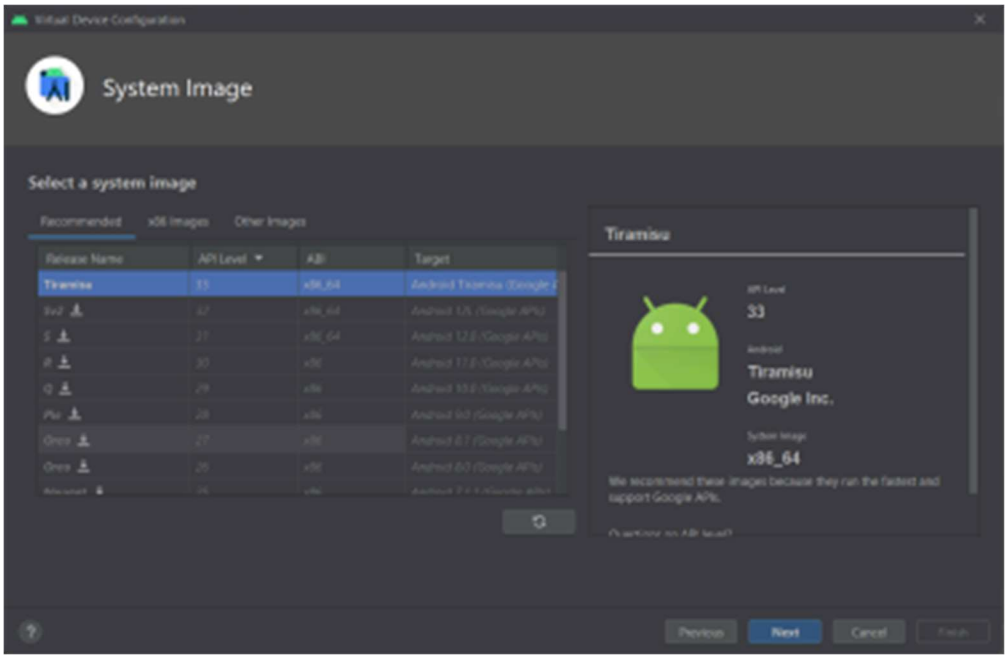


Step 8.2: Choose your device definition and click on Next.



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.

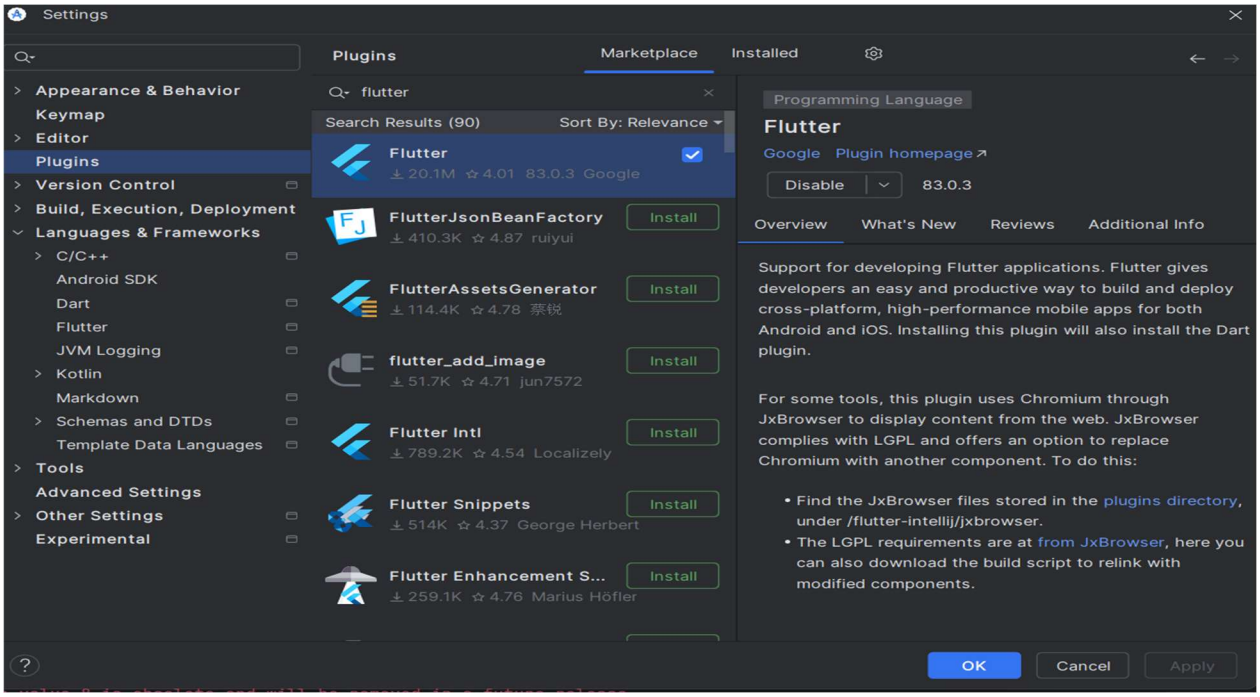


Step 8.5: Last, click on the icon pointed into the red color rectangle. The Android emulator
displayed as below screen.

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

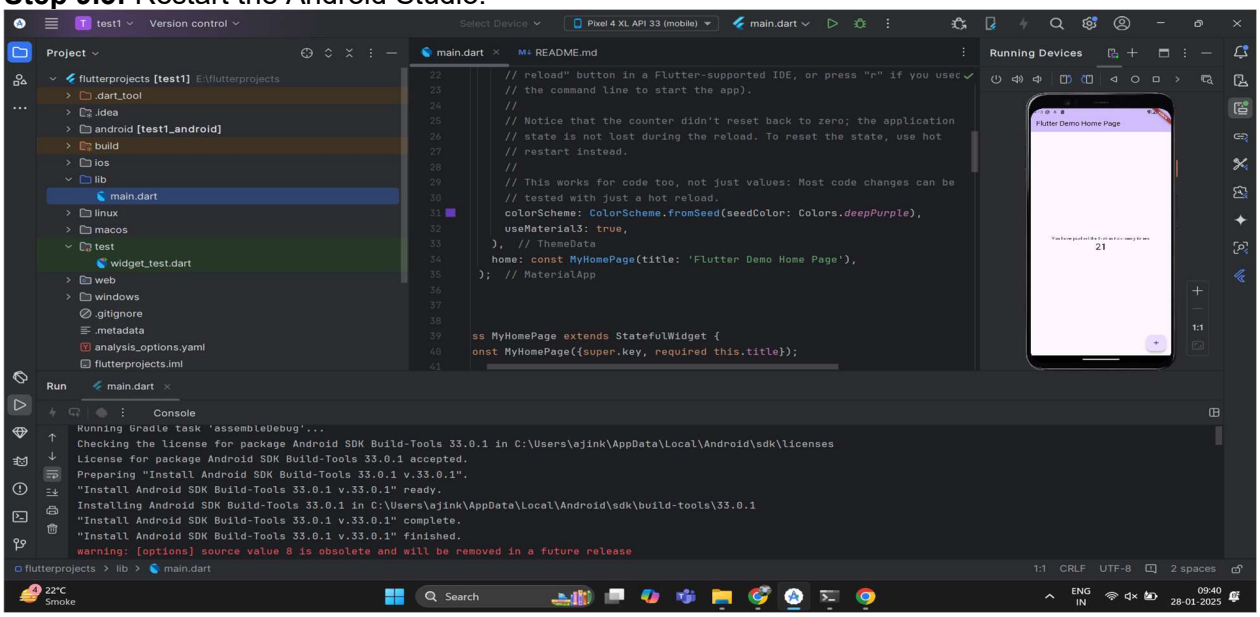
Flutter application in the Android Studio itself. Do the following steps to install these plugins.

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.



Name: Varun Gupta
Class: D15B
Roll: 18

Experiment 02 : To design Flutter UI by including common widgets.

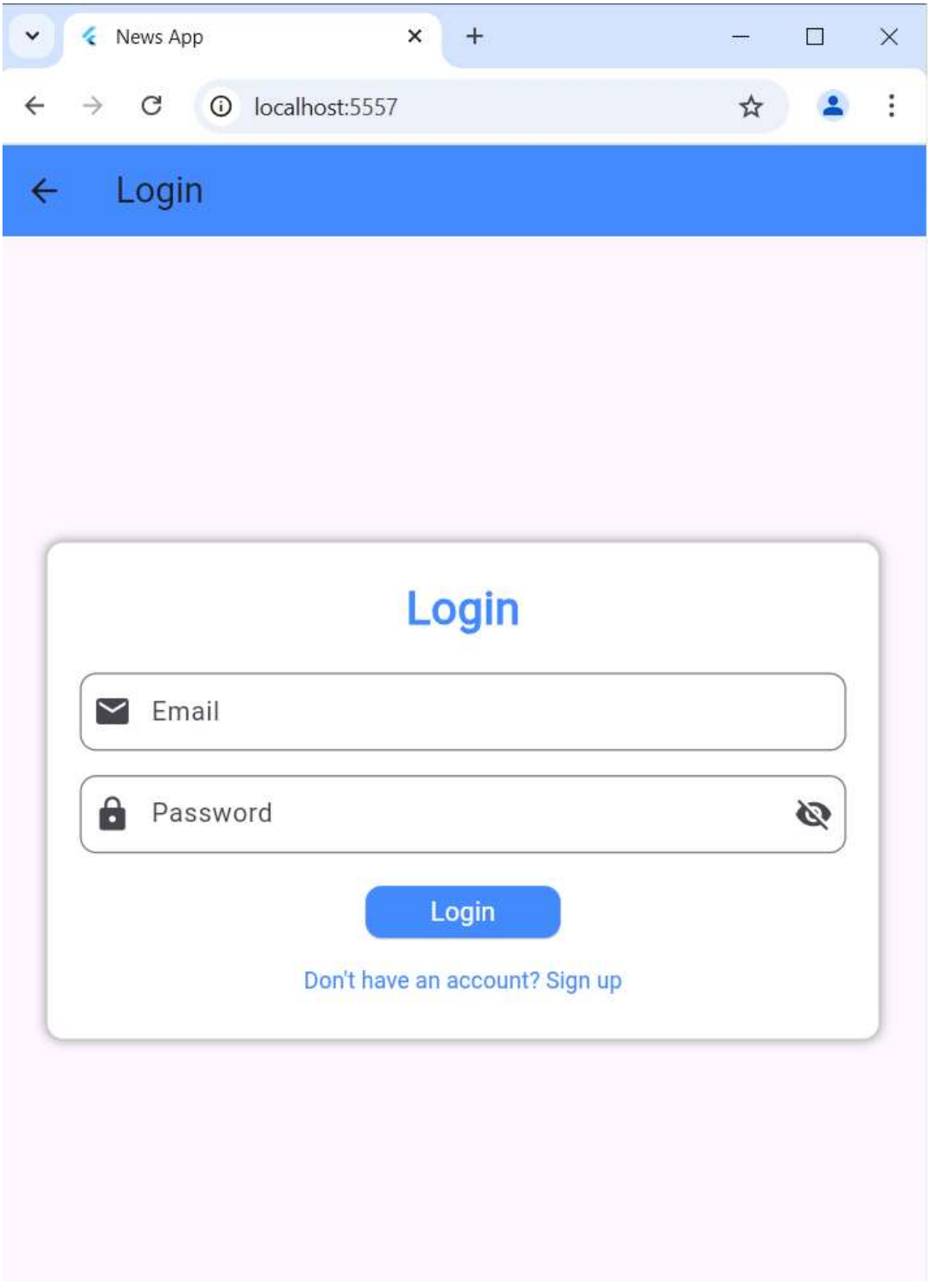
```
import 'package:flutter/material.dart';
import 'package:shared_preferences/shared_preferences.dart';
import 'home_screen.dart';
import 'signup_screen.dart';

class LoginPage extends StatefulWidget {
  @override
  _LoginPageState createState() => _LoginPageState();
}

class _LoginPageState extends State<LoginPage> {
  final TextEditingController emailController = TextEditingController();
  final TextEditingController passwordController =
TextEditingController();
  bool isPasswordVisible = false;
  String? errorMessage;

  Future<void> _login() async {
    SharedPreferences prefs = await SharedPreferences.getInstance();
    String? savedEmail = prefs.getString('user_email');
    String? savedPassword = prefs.getString('user_password');

    if (emailController.text == savedEmail &&
        passwordController.text == savedPassword) {
      await prefs.setBool('is_logged_in', true);
      Navigator.pushReplacement(
        context,
        MaterialPageRoute(builder: (context) => NewsHomePage()),
      );
    } else {
      setState(() {
        errorMessage = 'Invalid credentials';
      });
    }
  }
```

```
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text("Login"),
        backgroundColor: Colors.blueAccent,
        leading: IconButton(
          icon: Icon(Icons.arrow_back),
          onPressed: () => Navigator.pop(context),
        ),
      ),
      body: Center(
        child: Container(
          padding: EdgeInsets.all(20),
          margin: EdgeInsets.symmetric(horizontal: 30),
          decoration: BoxDecoration(
            color: Colors.white,
            borderRadius: BorderRadius.circular(10),
            boxShadow: [BoxShadow(color: Colors.grey, blurRadius: 5,
spreadRadius: 1)],
          ),
          child: Column(
            mainAxisSize: MainAxisSize.min,
            children: [
              Text("Login", style: TextStyle(fontSize: 28, fontWeight:
FontWeight.bold, color: Colors.blueAccent)),
              SizedBox(height: 20),
              TextField(
                controller: emailController,
                decoration: InputDecoration(
                  labelText: "Email",
                  border: OutlineInputBorder(borderRadius:
BorderRadius.circular(10)),
                  prefixIcon: Icon(Icons.email),
                ),
              ),
              SizedBox(height: 15),
              TextField(
                controller: passwordController,
```

```
                obscureText: !isPasswordVisible,
                decoration: InputDecoration(
                  labelText: "Password",
                  border: OutlineInputBorder(borderRadius:
BorderRadius.circular(10)),
                  prefixIcon: Icon(Icons.lock),
                  suffixIcon: IconButton(
                    icon: Icon(isPasswordVisible ? Icons.visibility :
Icons.visibility_off),
                    onPressed: () => setState(() => isPasswordVisible =
!isPasswordVisible),
                  ),
                ),
              ),
              if (errorMessage != null) Padding(
                padding: EdgeInsets.only(top: 10),
                child: Text(errorMessage!, style: TextStyle(color:
Colors.red)),
              ),
              SizedBox(height: 20),
              ElevatedButton(
                onPressed: _login,
                style: ElevatedButton.styleFrom(
                  backgroundColor: Colors.blueAccent,
                  shape: RoundedRectangleBorder(borderRadius:
BorderRadius.circular(10)),
                  padding: EdgeInsets.symmetric(horizontal: 40, vertical:
12),
                ),
                child: Text("Login", style: TextStyle(fontSize: 16,
color: Colors.white)),
              ),
              SizedBox(height: 10),
              TextButton(
                onPressed: () => Navigator.push(context,
MaterialPageRoute(builder: (context) => SignupPage())),
                child: Text("Don't have an account? Sign up", style:
TextStyle(color: Colors.blueAccent)),
              ),
            ],
          ),
        ),
```

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

OUTPUT :


News App
localhost:5557
Login
Login
Email
Password
Login
Don't have an account? Sign up

Usage of Widgets:

● Widgets like Row, Column, SizedBox, ElevatedButton, and Align are used to structure the UI.

● The SafeArea widget ensures that content is displayed within the safe area of the screen.

● The SingleChildScrollView widget allows scrolling when the content overflows the screen

List of Widgets

Flutter Scaffold
Flutter Container
Flutter Row & Column
Flutter Text
Flutter TextField
Flutter Buttons
Flutter Stack
Flutter Forms
Flutter AlertDialog
Flutter Icons
Flutter Images
Flutter Card
Flutter Tabbar
Flutter Drawer
Flutter Lists
Flutter GridView
Flutter Toast
Flutter Checkbox
Flutter Radio Button
Flutter Progress Bar
Flutter Snackbar
Flutter Tooltip
Flutter Slider
Flutter Switch
Flutter Charts
Bottom Navigation Bar
Flutter Themes
Flutter Table
Flutter Calendar
Flutter Animation

Name: Varun Gupta

Class: D15B
Roll: 18

Experiment 03: Including Icons, Images, and Fonts in a Flutter App

Introduction

Flutter allows seamless integration of icons, images, and custom fonts to enhance the visual appeal of mobile applications. In this document, we will explore how to include and use these assets effectively.

1. Including Icons in Flutter

Flutter provides built-in icons via the `Icons` class and allows the use of custom icons through the `pubspec.yaml` file.

1.1 Using Built-in Icons

Flutter provides an extensive set of material icons that can be used as follows:

```
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("Flutter Icons")),
        body: Center(
          child: Icon(
            Icons.favorite,
```

```
            color: Colors.red,
            size: 50.0,
          ),
        ),
      ),
    );
  }
}
```

1.2 Using Custom Icons

To use custom icons, first, download or generate an icon font using FlutterIcon and add it to the `pubspec.yaml` file:

```
flutter:
  fonts:
    - family: CustomIcons
      fonts:
        - asset: assets/fonts/custom_icons.ttf
```

Use it in your app:

```
Icon(IconData(0xe900, fontFamily: 'CustomIcons'))
```

2. Including Images in Flutter

Flutter supports various ways to include images, such as from the assets folder, network URLs, and memory.

2.1 Adding Image Assets

1. Place images inside the `assets/images/` directory.
2. Declare them in `pubspec.yaml`:

```
flutter:
  assets:
    - assets/images/sample.png
```

3. Use them in your app:

```
Image.asset('assets/images/sample.png', width: 200, height: 200)
```

2.2 Using Network Images

Load images directly from the internet:

Image.network('https://example.com/sample.jpg')

3. Including Custom Fonts in Flutter

Custom fonts can enhance the UI by providing unique typography.

3.1 Adding Custom Fonts

1. Place font files in `assets/fonts/`.
2. Declare them in `pubspec.yaml`:

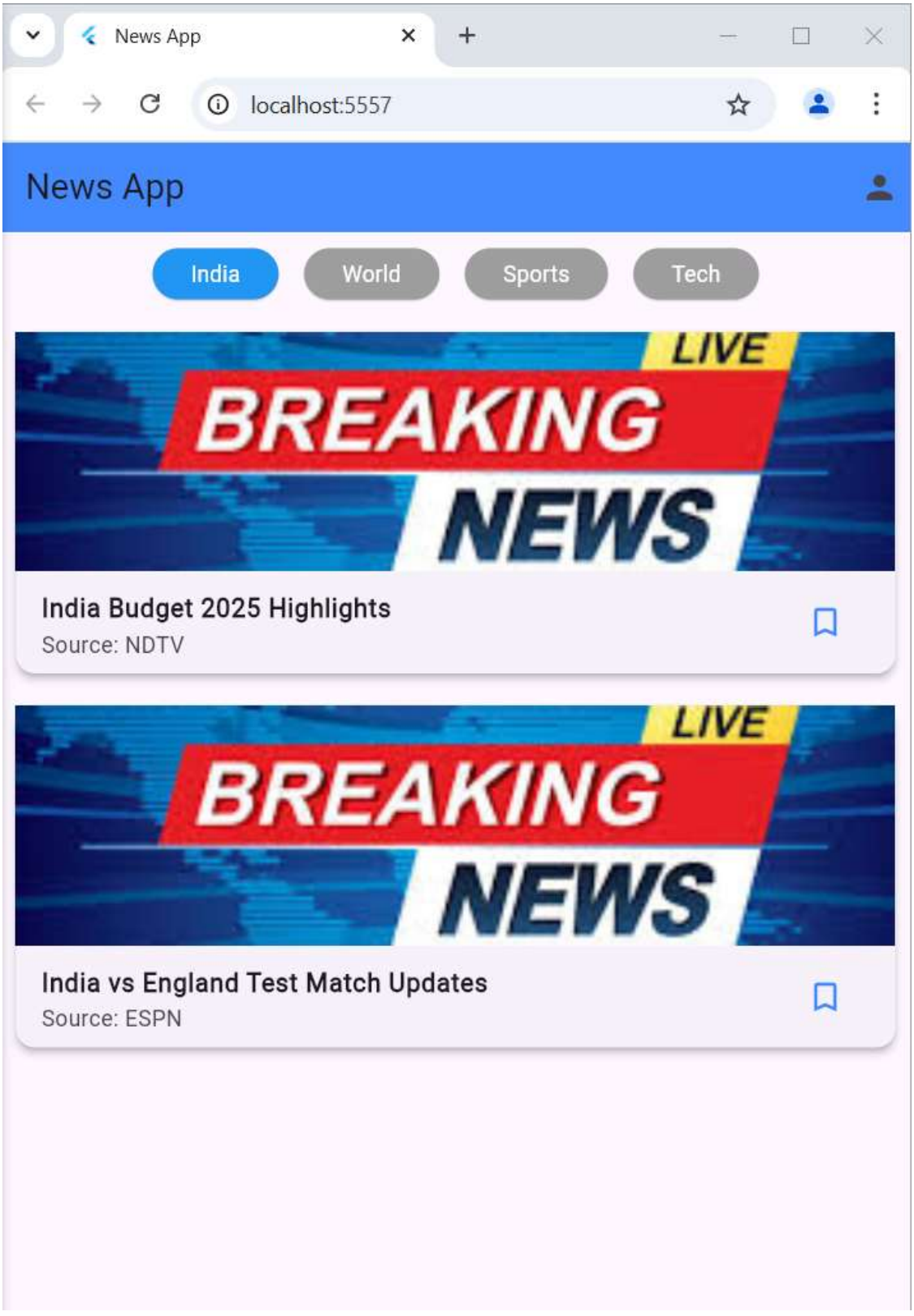
```
flutter:
  fonts:
    - family: CustomFont
      fonts:
        - asset: assets/fonts/CustomFont-Regular.ttf
```

3.2 Using Custom Fonts in the App

Apply the font to text widgets:

```
Text(
  'Hello, Flutter!',
  style: TextStyle(fontFamily: 'CustomFont', fontSize: 24),
)
```

Output


News App
localhost:5557
News App
India
World
Sports
Tech
LIVE
BREAKING
NEWS
India Budget 2025 Highlights
Source: NDTV
LIVE
BREAKING
NEWS
India vs England Test Match Updates
Source: ESPN

Conclusion

This document covered the integration of icons, images, and fonts in a Flutter app. These elements significantly enhance UI design, making the app more engaging and visually appealing.

Name: Varun Gupta
Class: D15B
Roll: 18

Lab 04

Aim: To create an interactive Form using form widget

Theory

Form validation is an essential feature in mobile applications to ensure the correctness of user inputs before processing them. In Flutter, `Form` and `TextFormField` widgets are used to create forms with built-in validation capabilities.

Key Concepts:

1. GlobalKey: Used to uniquely identify the form and manage its state.
2. TextFormField: A widget that allows users to enter input and supports validation.
3. Validation Logic: Defines conditions to verify the correctness of input.
4. Submit Button: Triggers form validation and processes the input if valid.

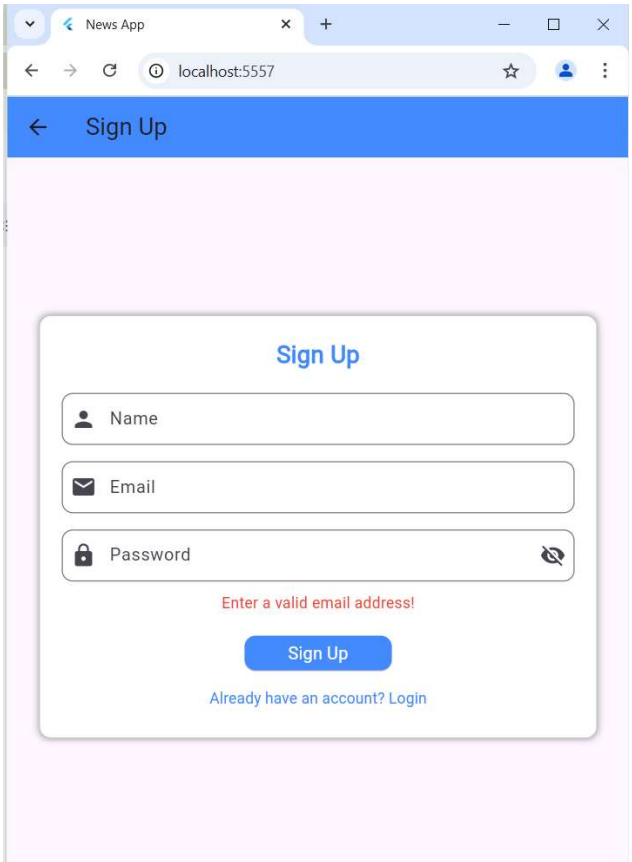
Code Implementation

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

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

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

```
}

class FormScreen extends StatefulWidget {
  @override
  _FormScreenState createState() => _FormScreenState();
}

class _FormScreenState extends State<FormScreen> {
  final _formKey = GlobalKey<FormState>();
  final TextEditingController _nameController = TextEditingController();

  void _submitForm() {
    if (_formKey.currentState!.validate()) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text('Form Submitted Successfully')),
      );
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Form Validation Demo')),
      body: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Form(
          key: _formKey,
          child: Column(
            children: [
              TextFormField(
                controller: _nameController,
                decoration: InputDecoration(labelText: 'Enter your name'),
                validator: (value) {
                  if (value == null || value.isEmpty) {
                    return 'Name cannot be empty';
                  }
                  return null;
```

```
            },
          ),
          SizedBox(height: 20),
          ElevatedButton(
            onPressed: _submitForm,
            child: Text('Submit'),
          ),
        ],
      ),
    ),
  ),
 );
 }
}
```

Output



Conclusion

Form validation in Flutter can be efficiently managed using the `GlobalKey`, `TextFormField`, and validation functions. This ensures user inputs are validated before processing, improving app reliability.

Name: Varun Gupta
Class: D15B
Roll: 18

MAD Lab 5

Aim: Navigation, Routing, and Gestures in Flutter

Introduction

Navigation, routing, and gestures are essential for building interactive Flutter applications. Navigation allows users to move between screens, routing manages structured transitions, and gestures detect user interactions like taps, swipes, and long presses.

Routing and Navigation in Flutter

In Flutter, routing and navigation are essential for managing screen transitions in an application. Flutter uses a stack-based navigation system, where screens (also called routes) are managed using a Navigator widget.

Types of Navigation in Flutter

1. Imperative Navigation (Navigator API)

- Uses Navigator.push() and Navigator.pop()
- Follows a stack-based approach (LIFO - Last In, First Out)
- Best suited for simple applications

2. Declarative Navigation (Go Router, Auto Route)

- Uses URL-based navigation
- Ideal for large applications with deep linking
- Navigator and Routes in Flutter

The Navigator manages the stack of screens in a Flutter app. Each screen is called a Route, and the Navigator widget helps in transitioning between routes.

1. Navigation in Flutter

Flutter's Navigator widget manages a stack of screens. Below is an example of basic navigation between two screens.

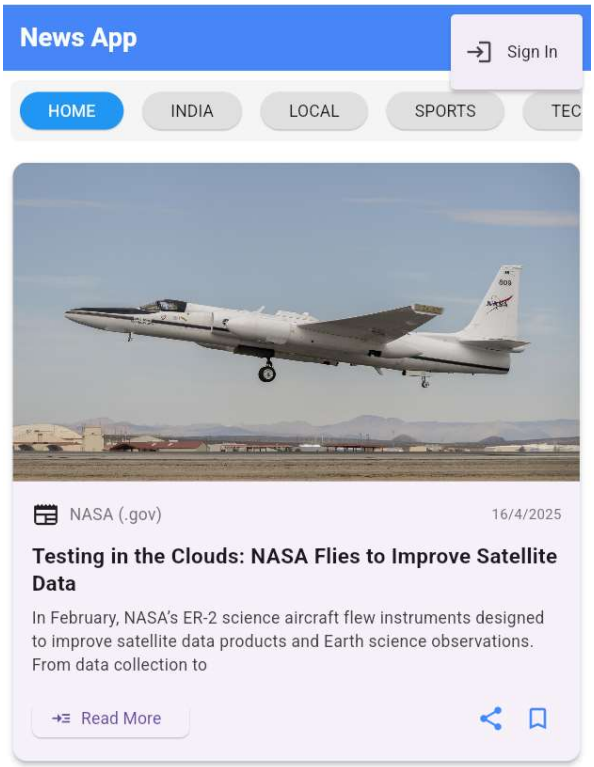
```
import 'package:flutter/material.dart';

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

class FirstScreen extends StatelessWidget { @override
    Widget build(BuildContext context) { return Scaffold(
            appBar: AppBar(title: Text('First Screen')), body: Center(
                child: ElevatedButton(
                    onPressed: () {
                        Navigator.push(
                            context, MaterialPageRoute(builder: (context) => SecondScreen()));
                    },
                    child: Text('Go to Second Screen'),
                ),
            ),
        );
    }
}

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

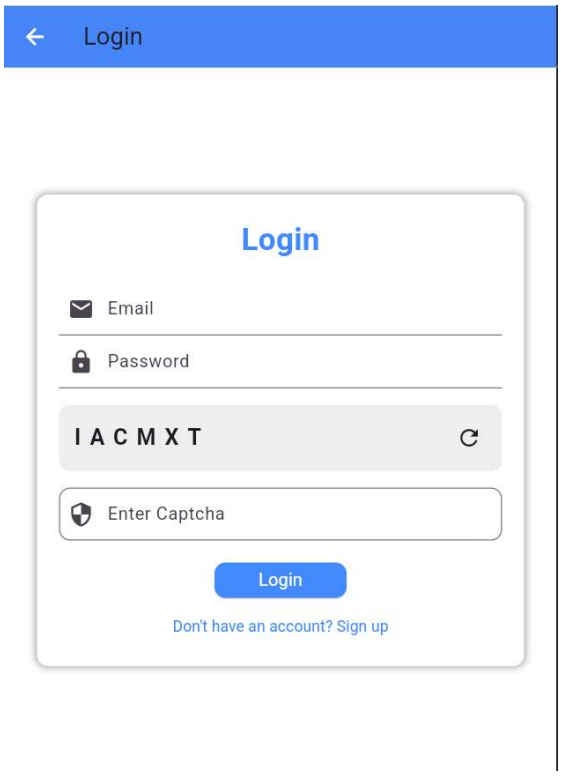
2. Gesture Detection in Flutter

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: GestureExample()));
}

class GestureExample extends StatefulWidget { @override
    _GestureExampleState createState() => _GestureExampleState();
}

class _GestureExampleState extends State<GestureExample> { String _message =
    "Tap or Swipe";

    @override
    Widget build(BuildContext context) { return Scaffold(
            appBar: AppBar(title: Text('Gesture Detector Example')), body:
            GestureDetector(
                onTap: () {
                    setState(() { _message = "Tapped!"; });
                },
                onDoubleTap: () {
                    setState(() { _message = "Double Tapped!"; });
                },
                onLongPress: () {
                    setState(() { _message = "Long Pressed!"; });
                },
                onHorizontalDragEnd: (details) { setState(() { _message =
                    "Swiped!"; });
                },
                child: Center(
                                    child: Text(_message, style: TextStyle(fontSize: 24, 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.

Experiment 06 : To set up Firebase with Flutter for Android Apps

Name: Varun Gupta

Class: D15B

Roll: 18

Firebase is a great backend solution for anyone that wants to use authentication, databases, cloud functions, ads, and countless other features within an app.

In this article, we will create a Firebase project for iOS and Android platforms using Flutter.

Prerequisites

To complete this tutorial, you will need:

- A Google account to use Firebase.
- Developing for iOS will require XCode.
- To download and install Flutter.
- To download and install Android Studio and Visual Studio Code.
- It is recommended to install plugins for your code editor:
    - `Flutter` and `Dart` plugins installed for Android Studio.
    - `Flutter` extension installed for Visual Studio Code.

Creating a New Flutter Project

This tutorial will require the creation of an example Flutter app.
Once you have your environment set up for Flutter, you can run the following to create a new application:

```
flutter create flutterfirebaseexample
```

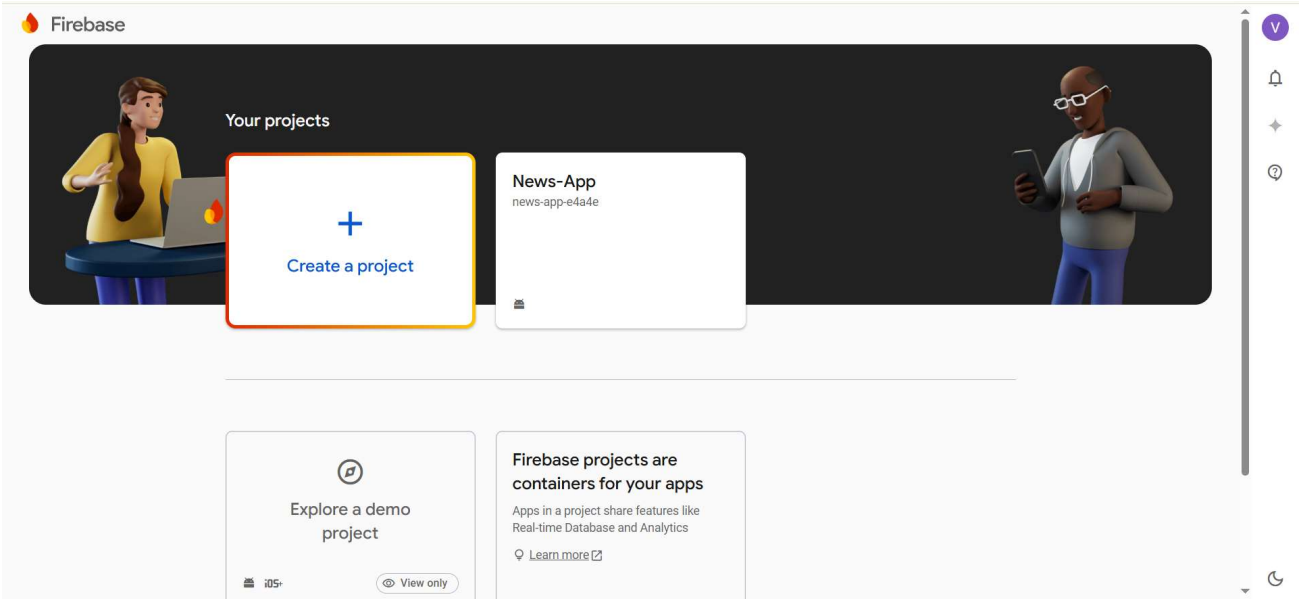
Navigate to the new project directory:

```
cd flutterfirebaseexample
```

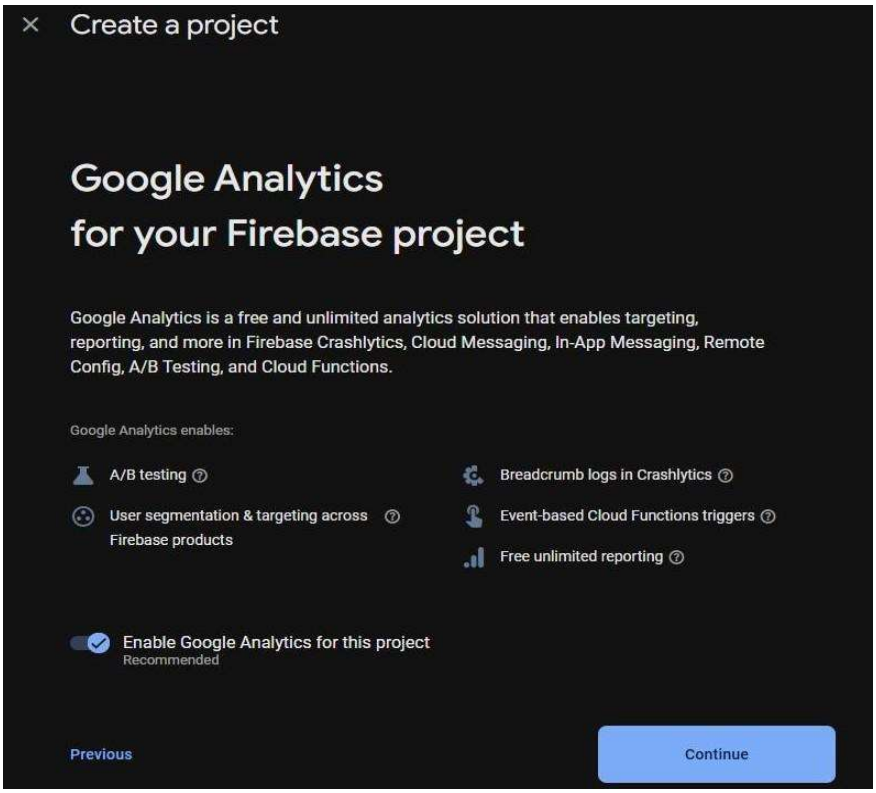
Using `flutter create` will produce a demo application that will display the number of times a button is clicked.
Now that we've got a Flutter project up and running, we can add Firebase.

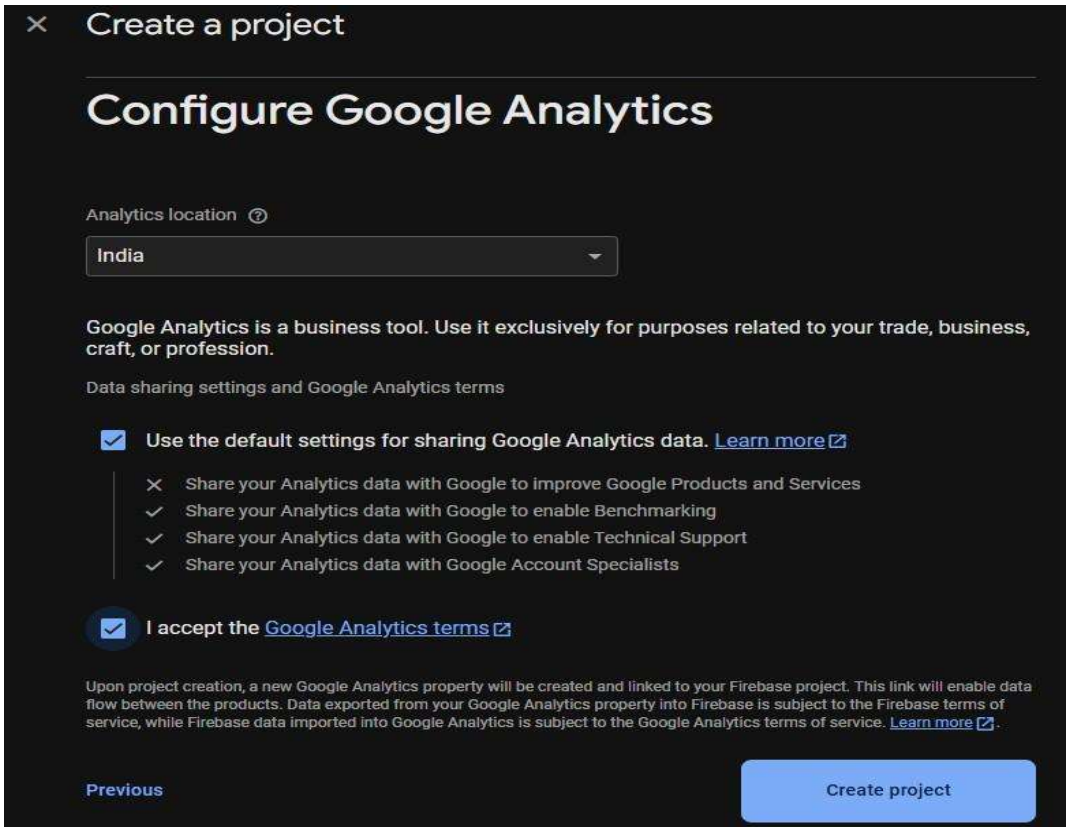
Creating a New Firebase Project

First, log in with your Google account to manage your Firebase projects. From within the Firebase dashboard, select the Create new project button and give it a name:



Next, we're given the option to enable Google Analytics. This tutorial will not require Google Analytics, but you can also choose to add it to your project.

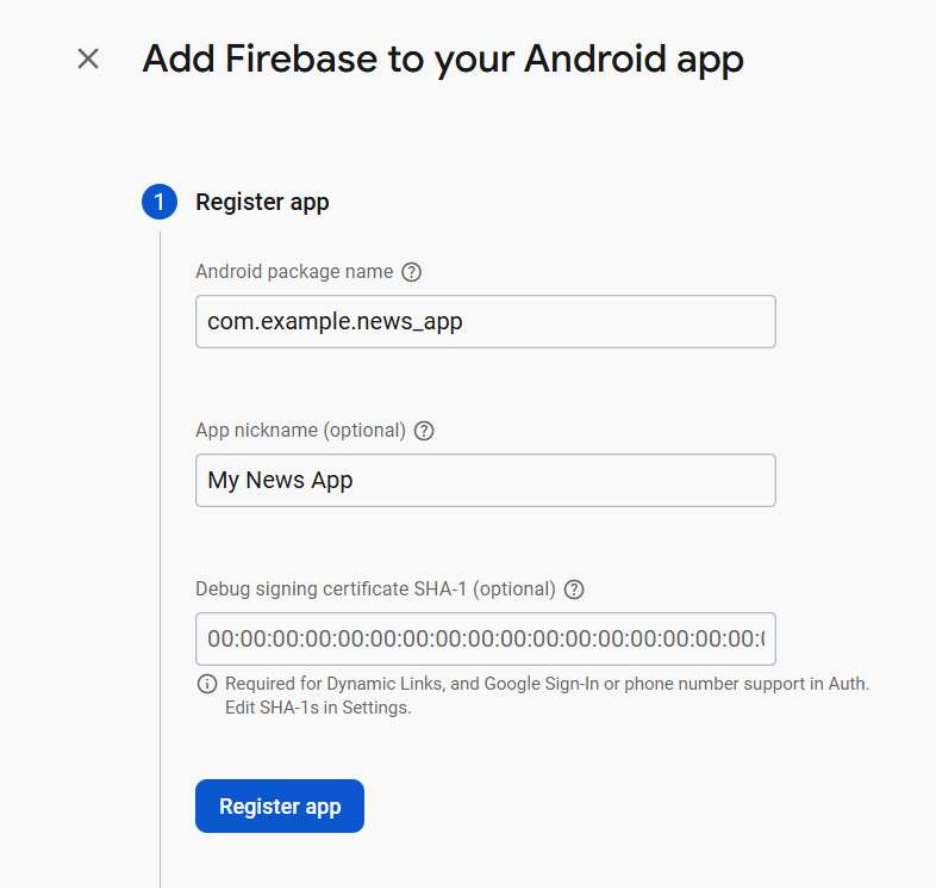




Adding Android support

Registering the App

In order to add Android support to our Flutter application, select the Android logo from the dashboard. This brings us to the following screen:



The most important thing here is to match up the Android package name that you choose here with the one inside of our application.

The structure consists of at least two segments. A common pattern is to use a domain name, a company name, and the application name:

```
com.example.flutterfirebaseexample
```

Once you've decided on a name, open `android/app/build.gradle` in your code editor and update the `applicationId` to match the Android package name:

android/app/build.gradle

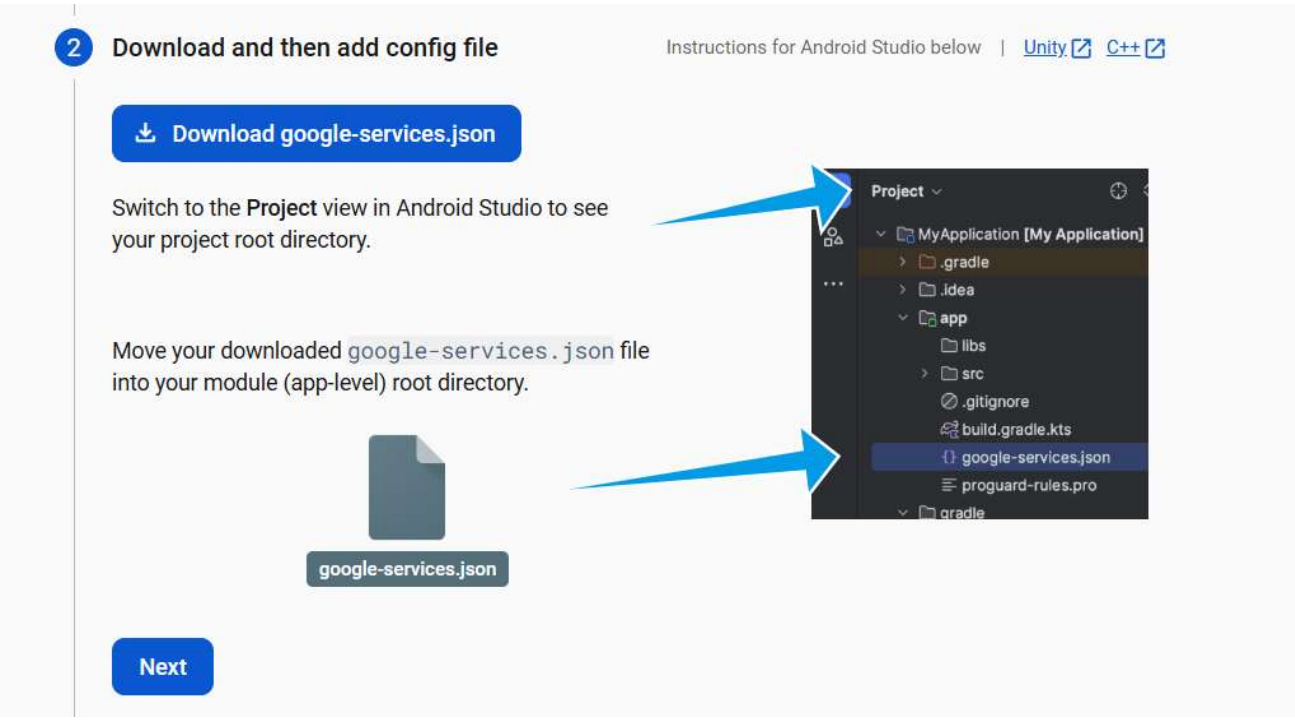
```
...
defaultConfig {
 // TODO: Specify your own unique Application ID
(https://developer.android.com/studio/build/application-id.html)
.  applicationId 'com.example.flutterfirebaseexample'
 ...
}
...
```

You can skip the app nickname and debug signing keys at this stage. Select Register app to continue.

Downloading the Config File

The next step is to add the Firebase configuration file into our Flutter project. This is important as it contains the API keys and other critical information for Firebase to use.

Select Download `google-services.json` from this page:



Next, move the `google-services.json` file to the `android/app` directory within the Flutter project.

Adding the Firebase SDK

We'll now need to update our Gradle configuration to include the Google Services plugin.

Open `android/build.gradle` in your code editor and modify it to include the following:

android/build.gradle

```
buildscript {
 repositories {
 // Check that you have the following line (if not, add it):
google() // Google's Maven repository
 }
 dependencies {
 ...
 // Add this line
 classpath 'com.google.gms:google-services:4.3.6'
 }
}

allprojects {
 ...
 repositories {
 // Check that you have the following line (if not, add it):
google() // Google's Maven repository
 ...
 }
}
```

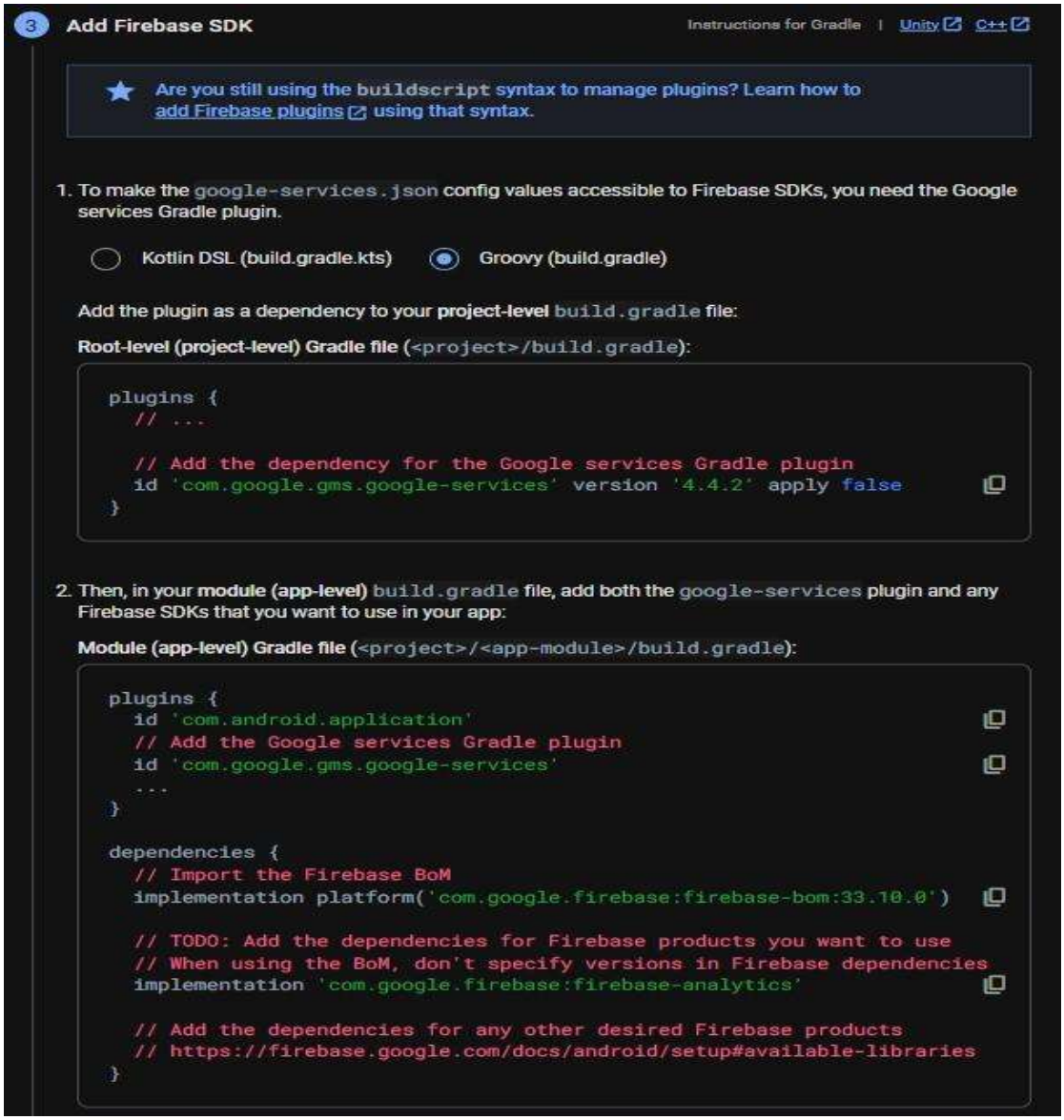
Finally, update the app level file at `android/app/build.gradle` to include the following:

android/app/build.gradle

```
apply plugin: 'com.android.application'
// Add this line
apply plugin: 'com.google.gms.google-services'

dependencies {
 // Import the Firebase BoM
 implementation platform('com.google.firebase:firebase-bom:28.0.0')

 // Add the dependencies for any other desired Firebase products  //
https://firebase.google.com/docs/android/setup#available-libraries }
```

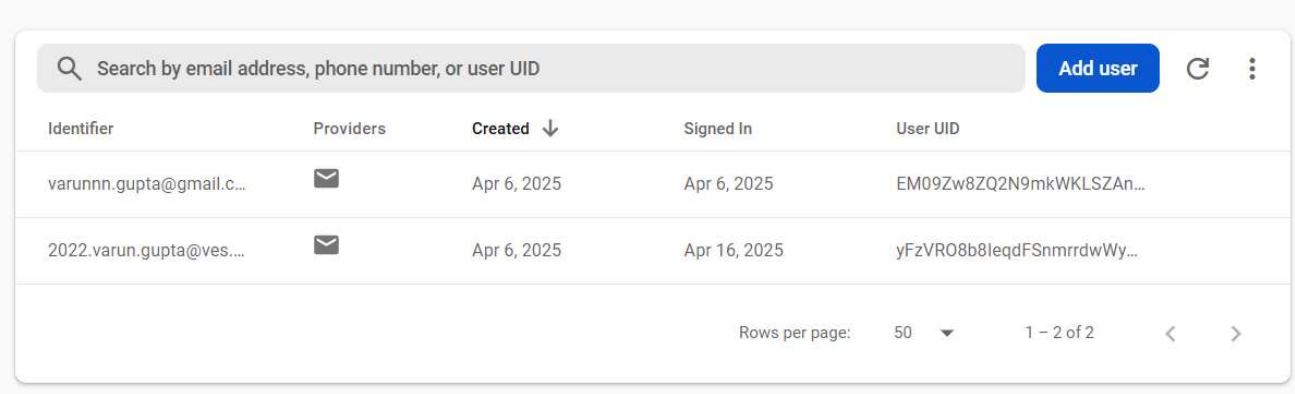
With this update, we're essentially applying the Google Services plugin as well as looking at how other Flutter Firebase plugins can be activated such as Analytics.

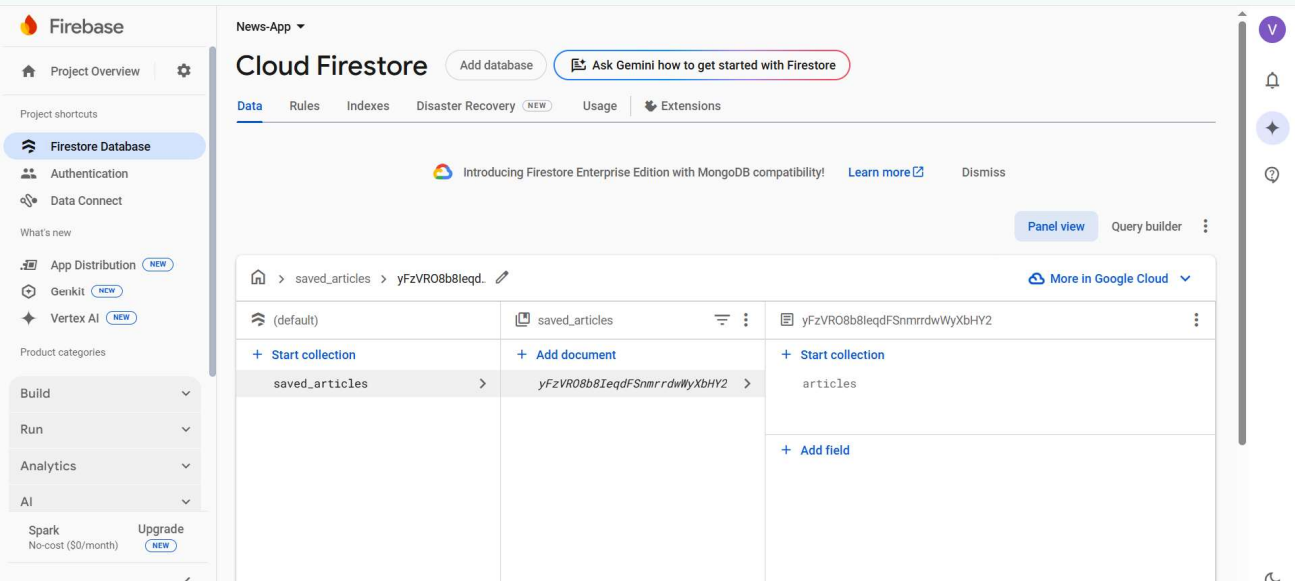
From here, run your application on an Android device or simulator. If everything has worked correctly, you should get the following message in the dashboard

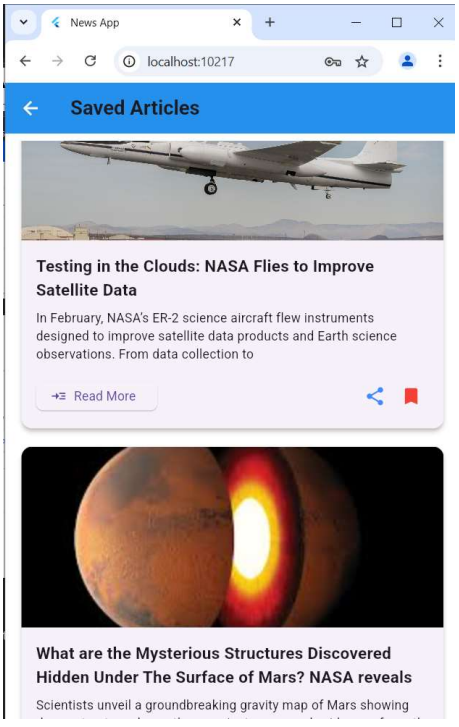


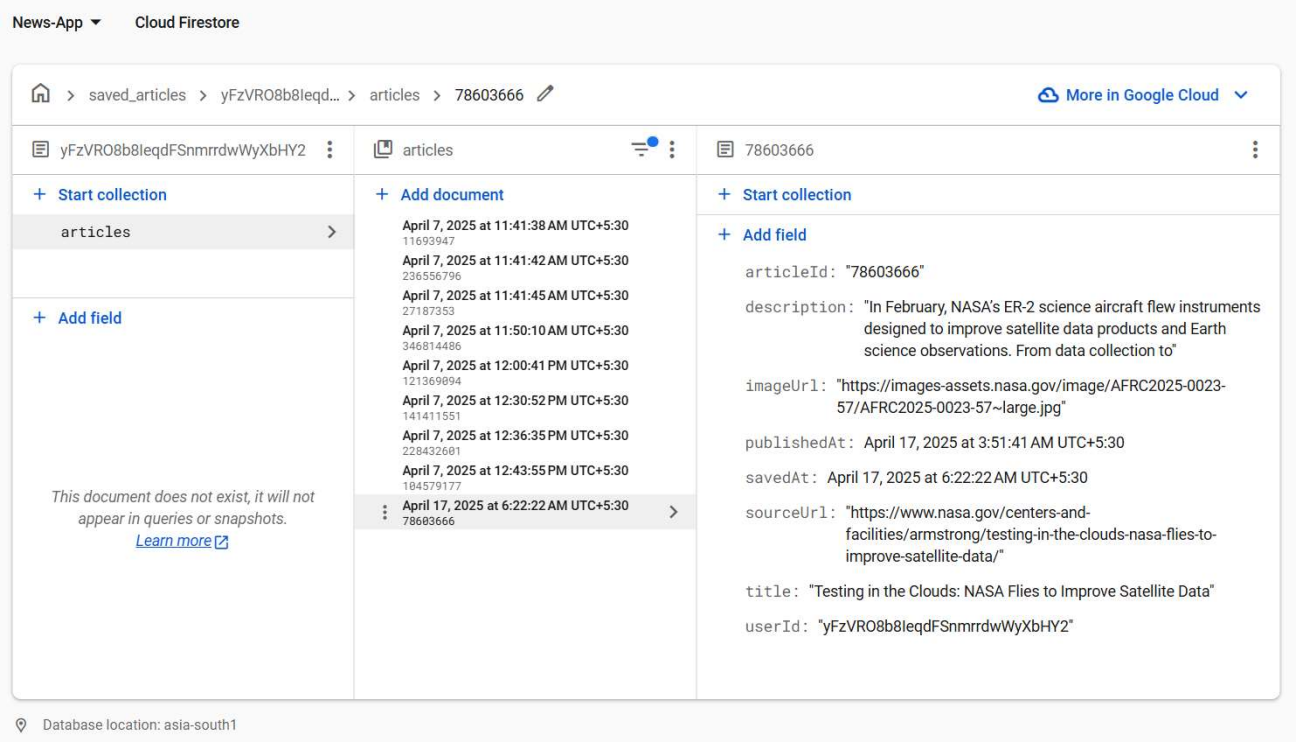
Be sure to move this file within Xcode to create the proper file references.
There are additional steps for installing the Firebase SDK and adding initialization code, but they are not necessary for this tutorial.

After Setup


Search by email address, phone number, or user UID
Add user
Identifier
Providers
Created
Signed In
User UID
varunnn.gupta@gmail.c...
Apr 6, 2025
Apr 6, 2025
EM09Zw8ZQ2N9mkWKLSZAn...
2022.varun.gupta@ves....
Apr 6, 2025
Apr 16, 2025
yFzVRO8b8IeqdFSnmrrdwWy...
Rows per page:
50
1 – 2 of 2


Firebase
Project Overview
Project shortcuts
Firestore Database
Authentication
Data Connect
What's new
App Distribution
Genkit
Vertex AI
Product categories
Build
Run
Analytics
AI
Spark
No-cost ($0/month)
Upgrade
News-App
Cloud Firestore
Add database
Ask Gemini how to get started with Firestore
Data
Rules
Indexes
Disaster Recovery
Usage
Extensions
Introducing Firestore Enterprise Edition with MongoDB compatibility!
Learn more
Dismiss
Panel view
Query builder
saved_articles
yFzVRO8b8IeqdFSnmrrdwWyXbHY2
More in Google Cloud
(default)
Start collection
saved_articles
Add document
articles
Add field


News App
localhost:10217
Saved Articles
Testing in the Clouds: NASA Flies to Improve Satellite Data
In February, NASA's ER-2 science aircraft flew instruments designed to improve satellite data products and Earth science observations. From data collection to
Read More
What are the Mysterious Structures Discovered Hidden Under The Surface of Mars? NASA reveals
Scientists unveil a groundbreaking gravity map of Mars showing



Conclusion

In this article, you learned how to set up and ready our Flutter applications to be used with Firebase.
Flutter has official support for Firebase with the FlutterFire set of libraries.

EXPERIMENT No. 7

Aim: To write meta data of your Blog PWA in a Web app manifest file to enable add to home screen feature

Theory:

Making 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 like 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.

Key Features of a PWA:

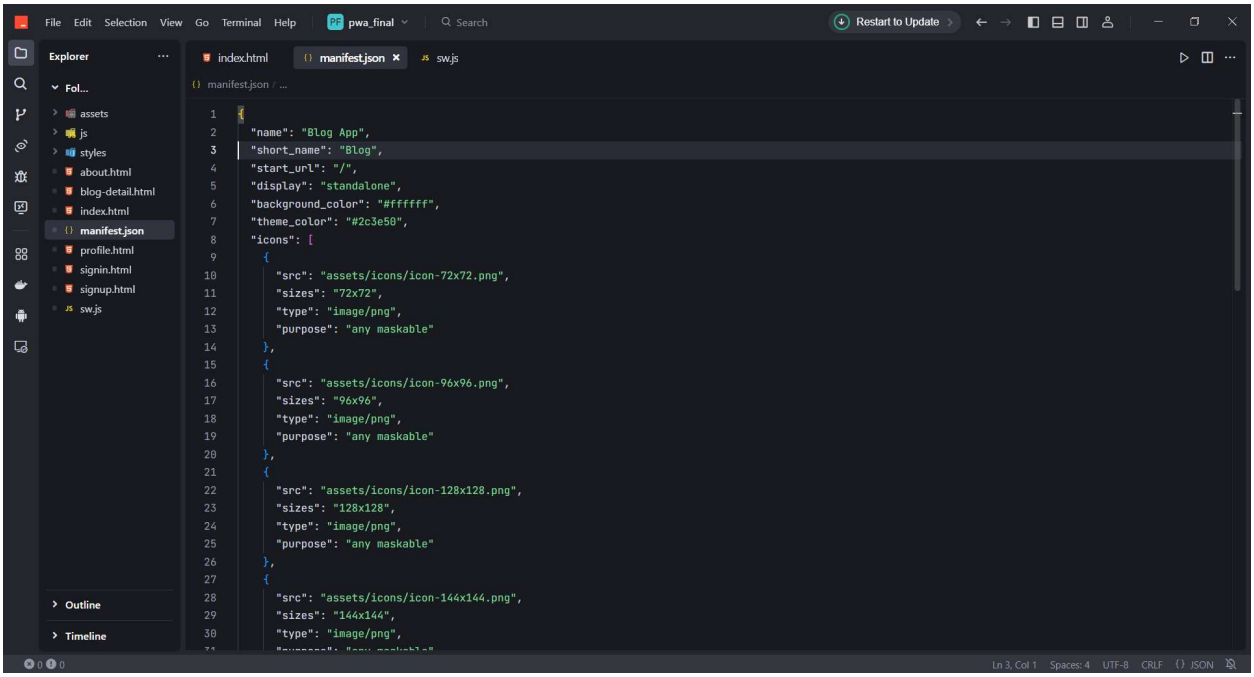
- Offline Availability: Allows users to access the app without an internet connection.
- Fast Performance: Uses caching strategies to reduce loading time.
- App-Like Interface: Can be installed on devices and launched in a standalone mode.
- Secure & Reliable: Served over HTTPS to prevent data tampering.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

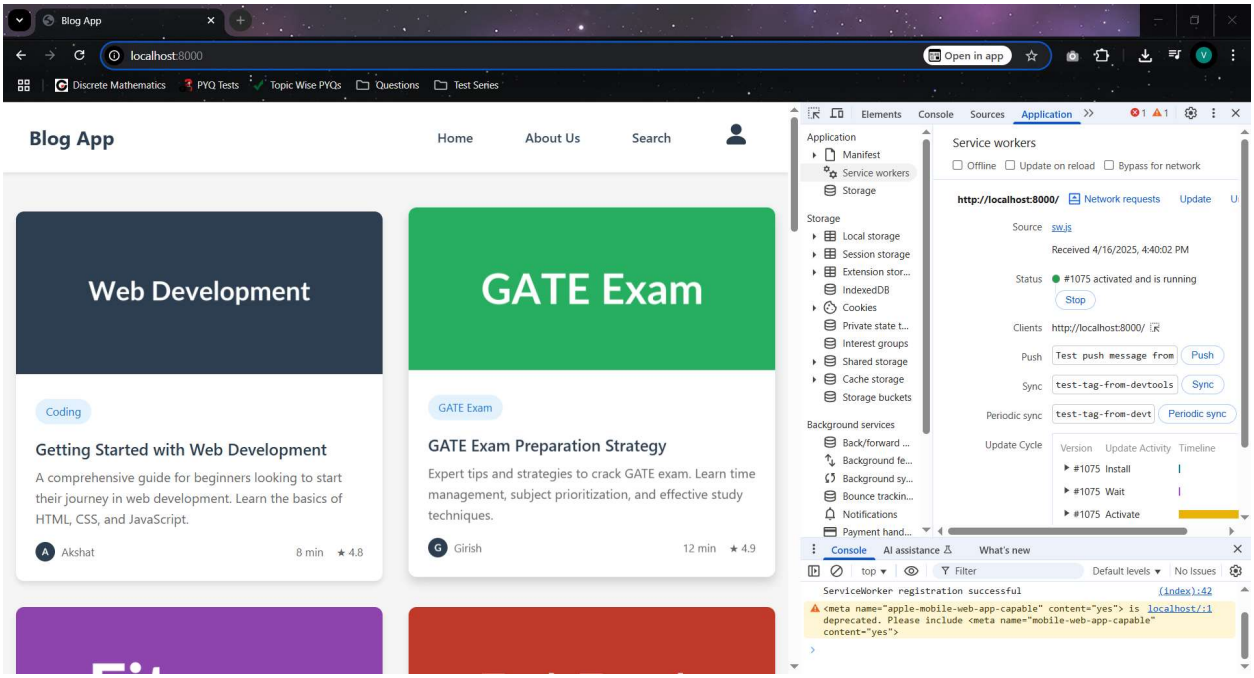
Steps to Make App a PWA

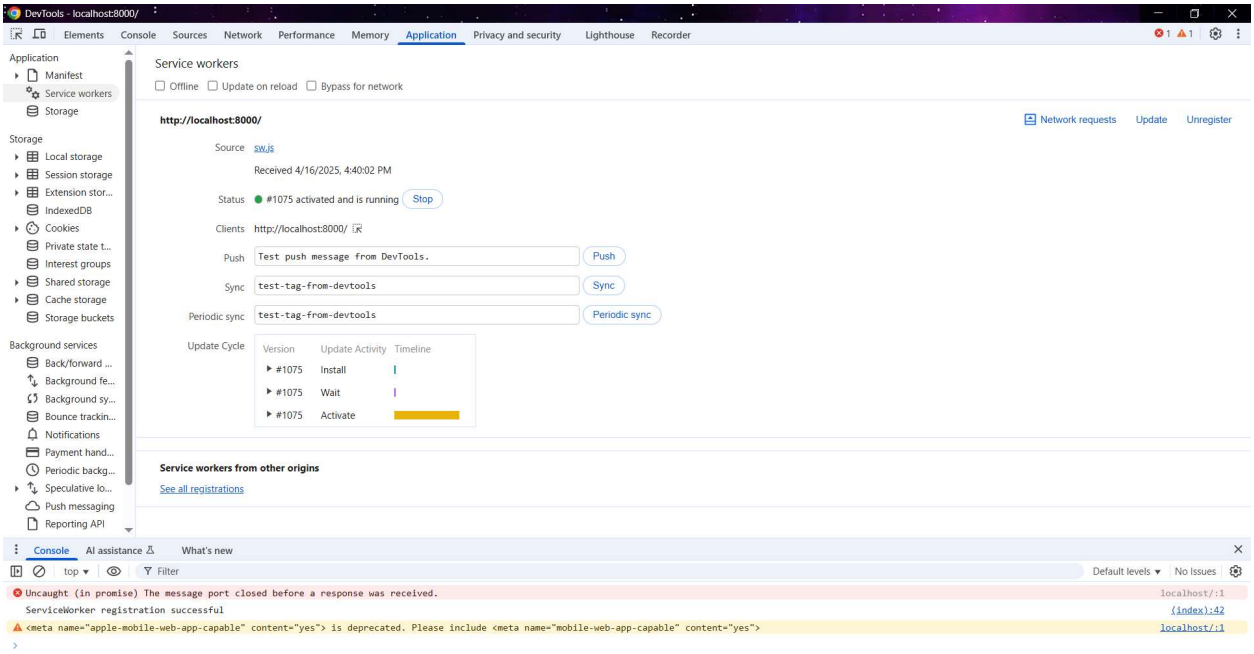
1. Add `manifest.json` File

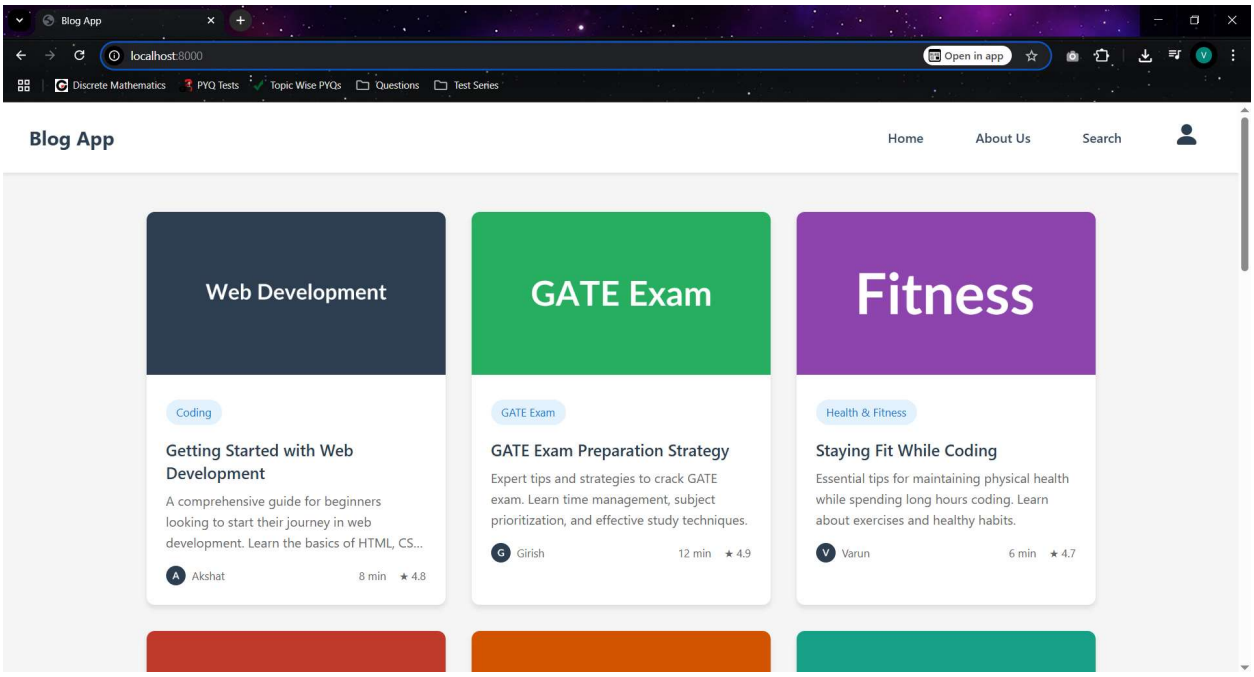
1. Go to the `public` folder in your project.
2. Create a new file named `manifest.json`.
3. Add details like the app name, icons, theme color, and display mode.
4. This file helps browsers recognize your app as installable.

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.

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 Blog PWA.

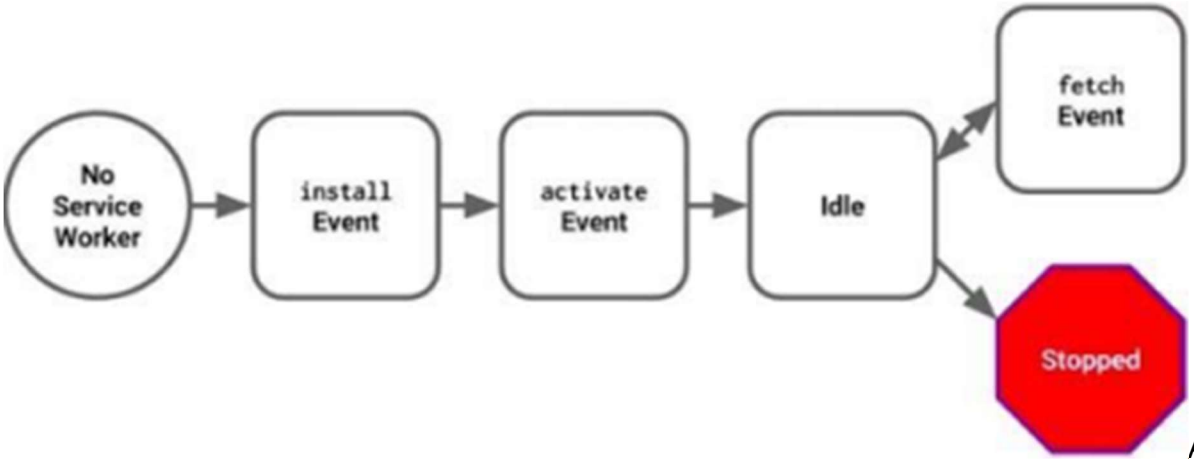
Theory:

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

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.

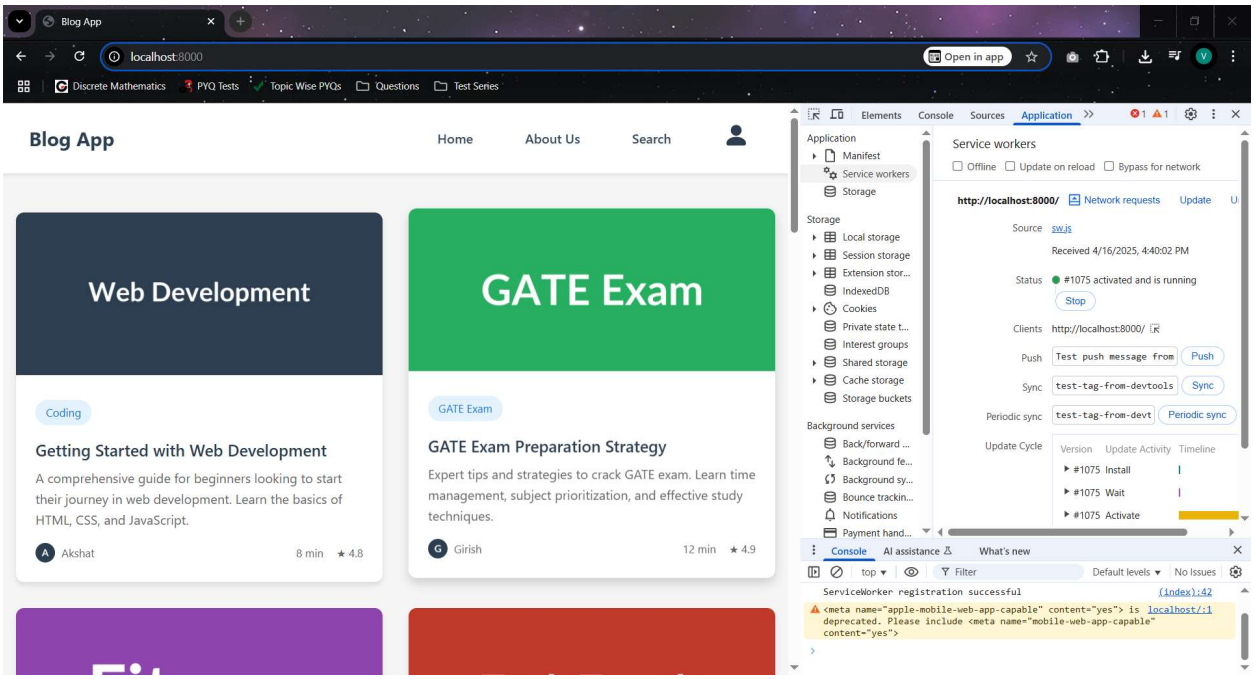
CODE :

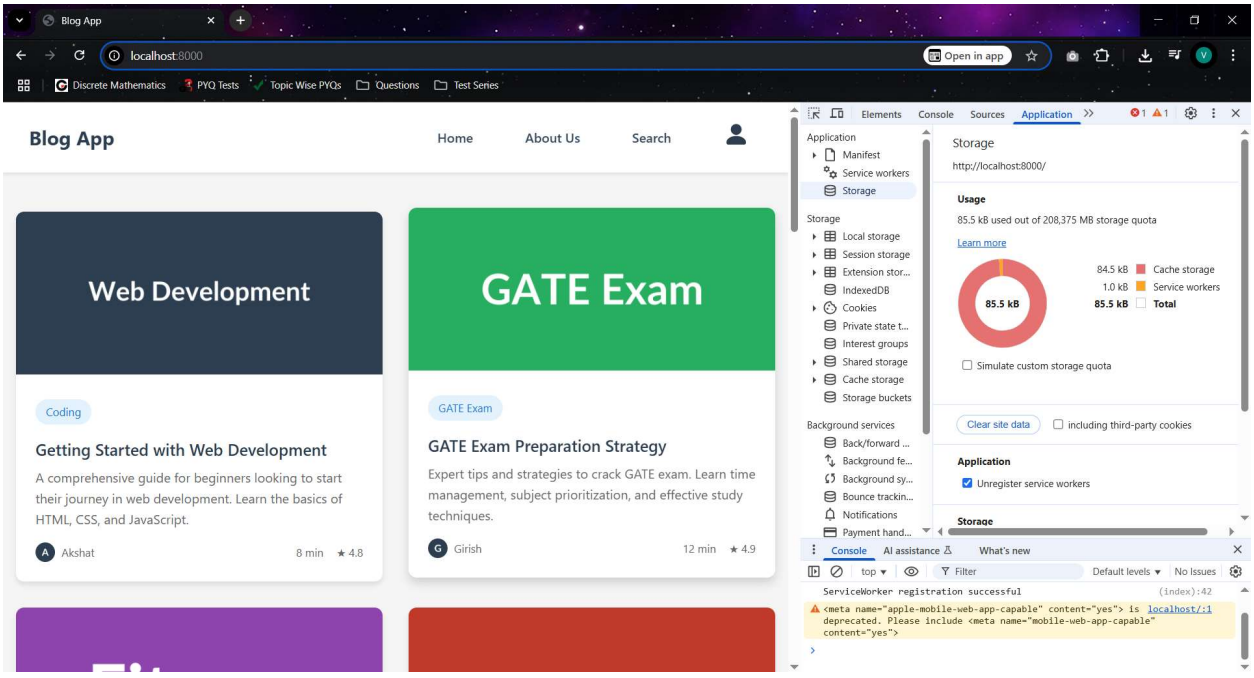
sw.js

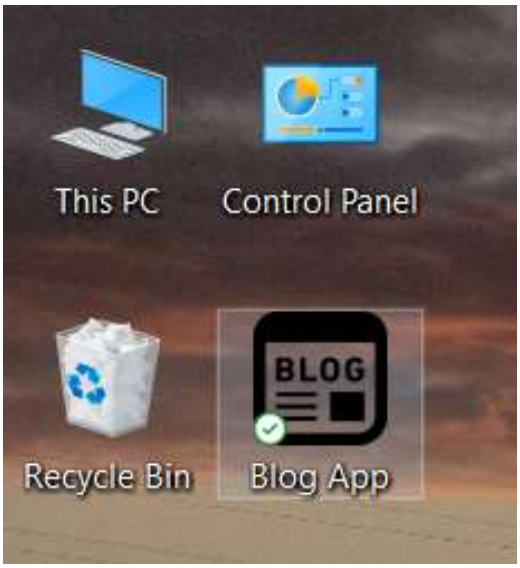
```
const CACHE_NAME = 'blog-app-v1';
const urlsToCache = [
  './',
  './index.html',
  './about.html',
  './blog-detail.html',
  './profile.html',
  './signin.html',
  './signup.html',
  './styles/main.css',
  './styles/navbar.css',
  './styles/blog-cards.css',
  './js/blogData.js',
  './js/features.js',
  './js/main.js',
  './assets/profile-icon.svg',
  './manifest.json'
];

self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then((cache) => cache.addAll(urlsToCache))
  );
});

self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request)
      .then((response) => response || fetch(event.request))
  );
});

self.addEventListener('activate', (event) => {
  event.waitUntil(
    caches.keys().then((cacheNames) => {
      return Promise.all(
        cacheNames.map((cacheName) => {
          if (cacheName !== CACHE_NAME) {
```

```
            return caches.delete(cacheName);
          }
        })
      );
    })
  );
});
```

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.

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

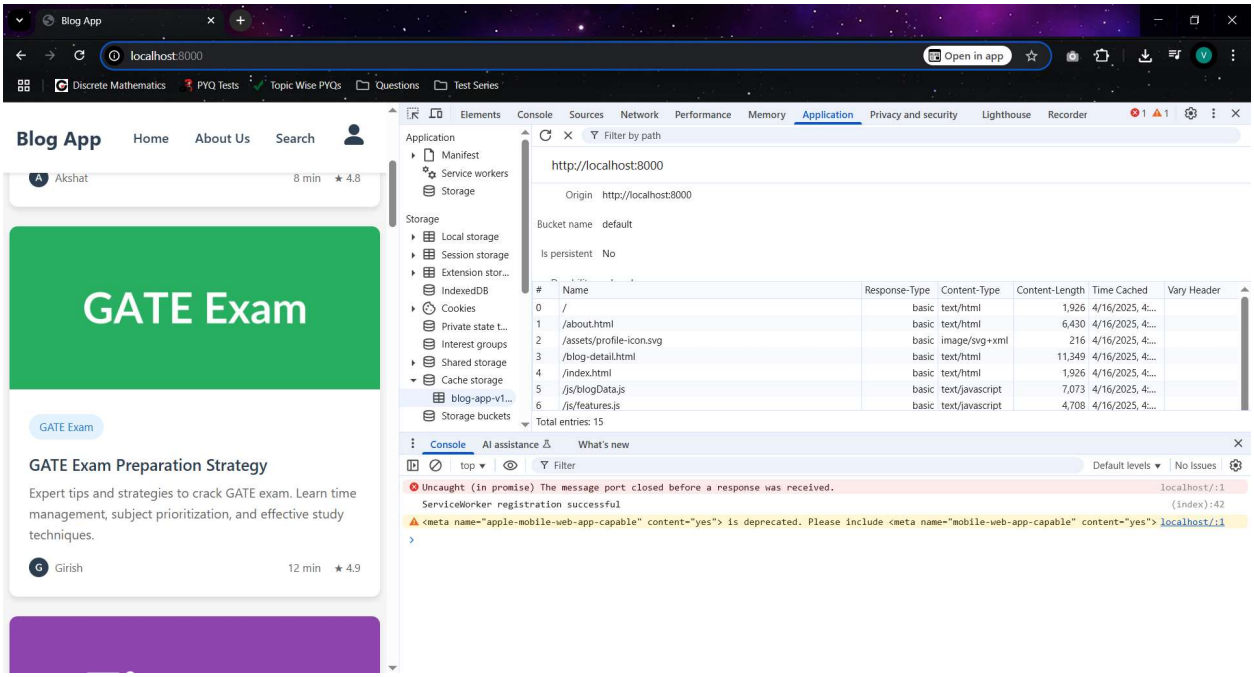
Progressive Web Apps (PWAs) are web applications that offer app-like experiences through modern web capabilities. One of the key components of a PWA is the service worker, which enables features like offline access, background sync, and push notifications. In this document, we will explore how to implement 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);
    })
  );
});
```



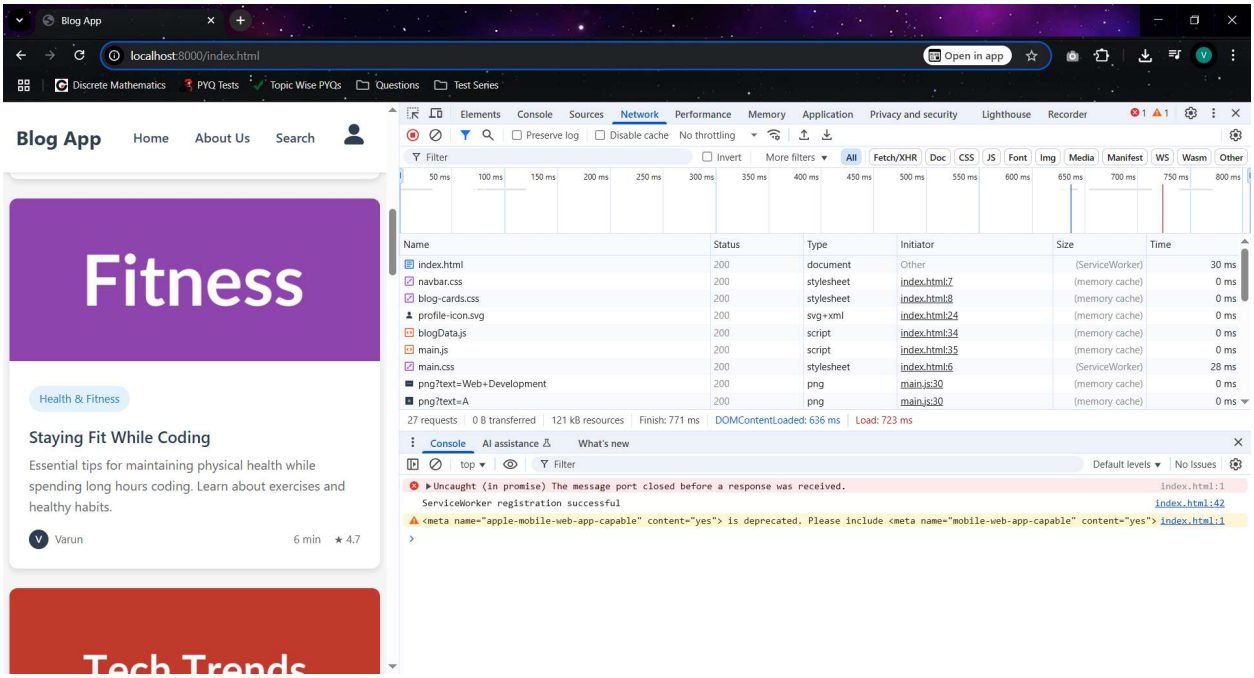
---

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)

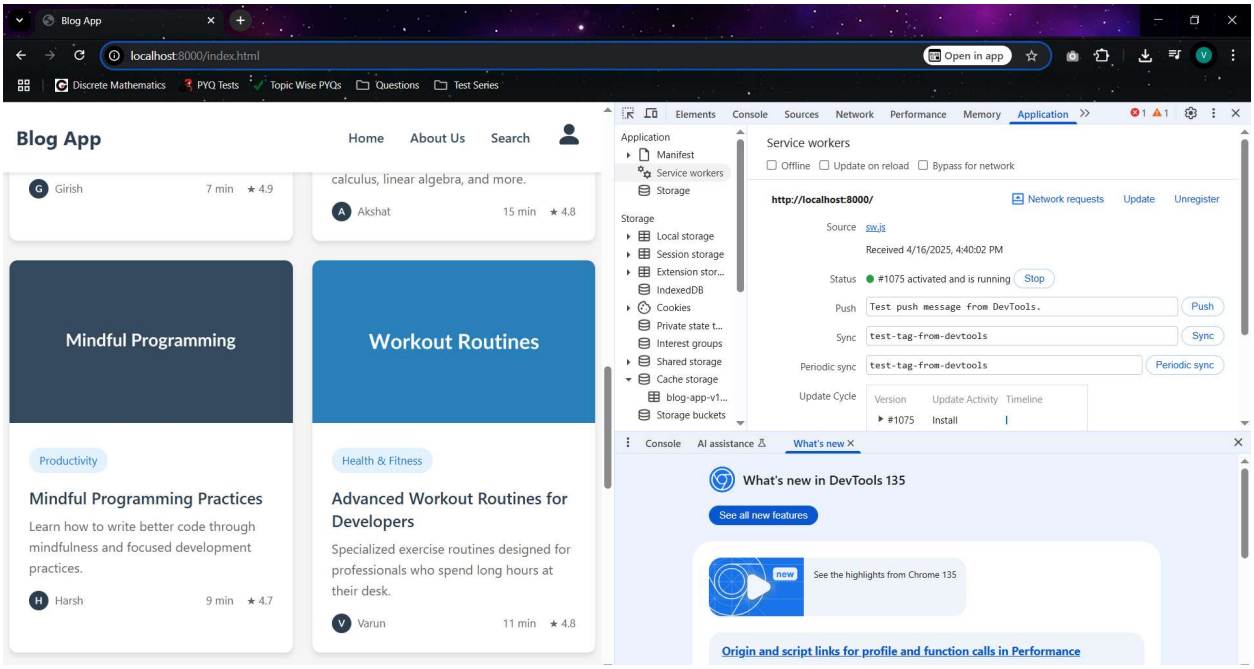
The sync event is used to defer actions until the user has stable connectivity. For an e-commerce app, you could use this to sync cart data or orders.

```
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.

Lab 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

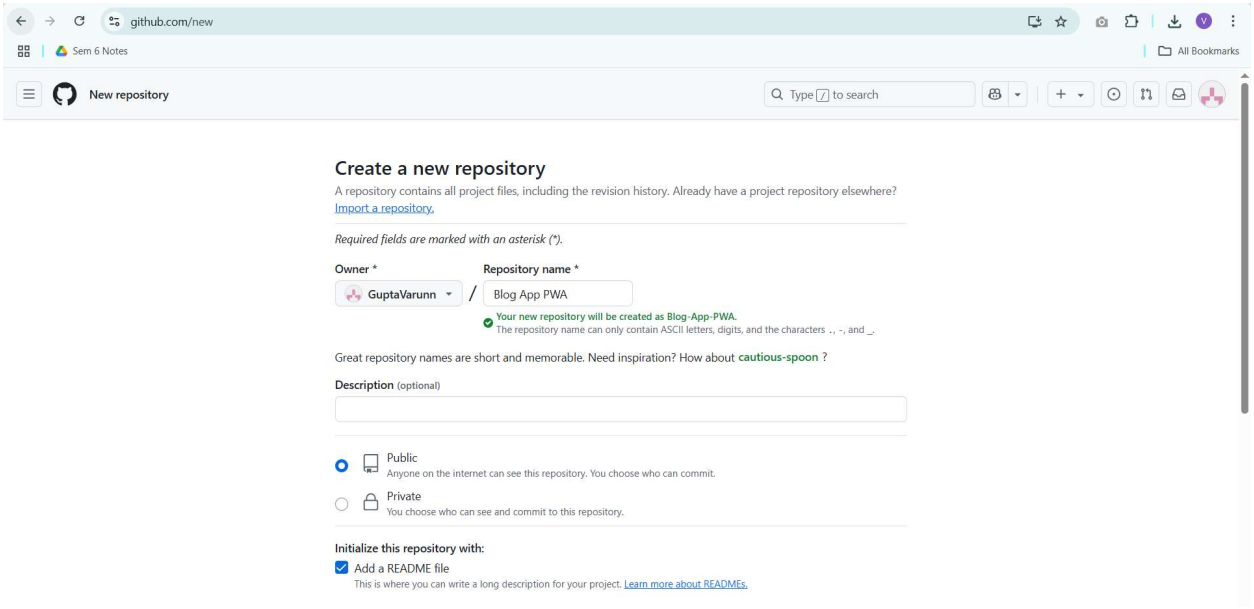
- A GitHub account
- A repository containing your static website (HTML, CSS, JS)
- Basic knowledge of Git and GitHub

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)

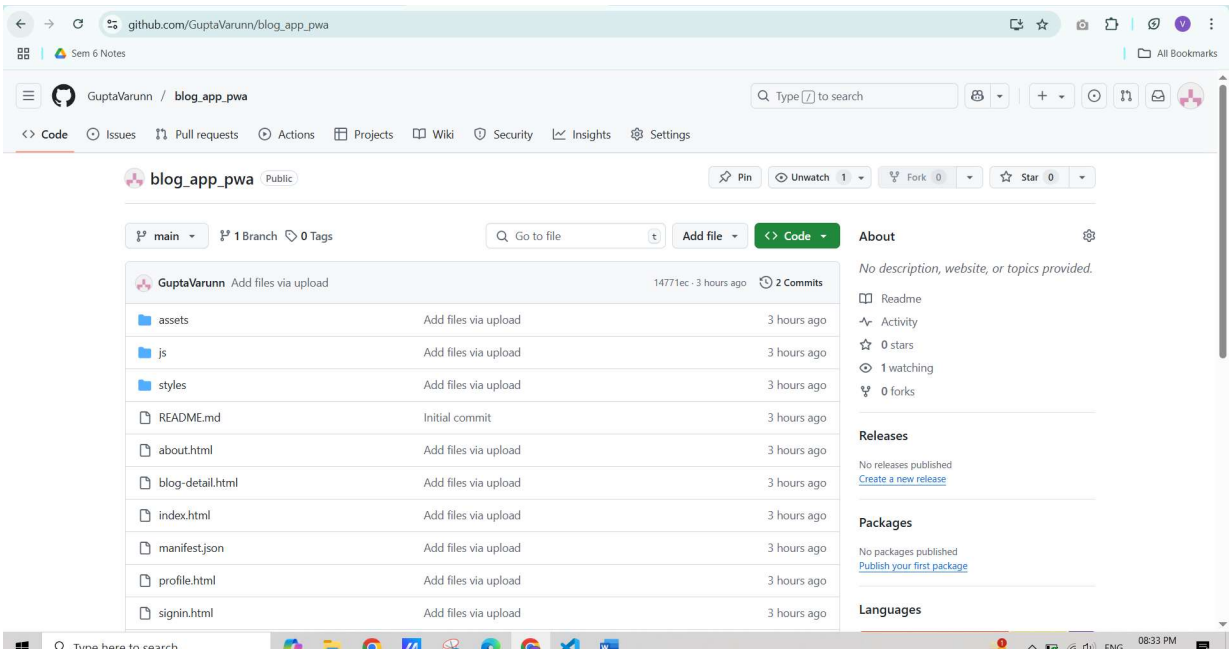


Step 2: Upload Your Website Files

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

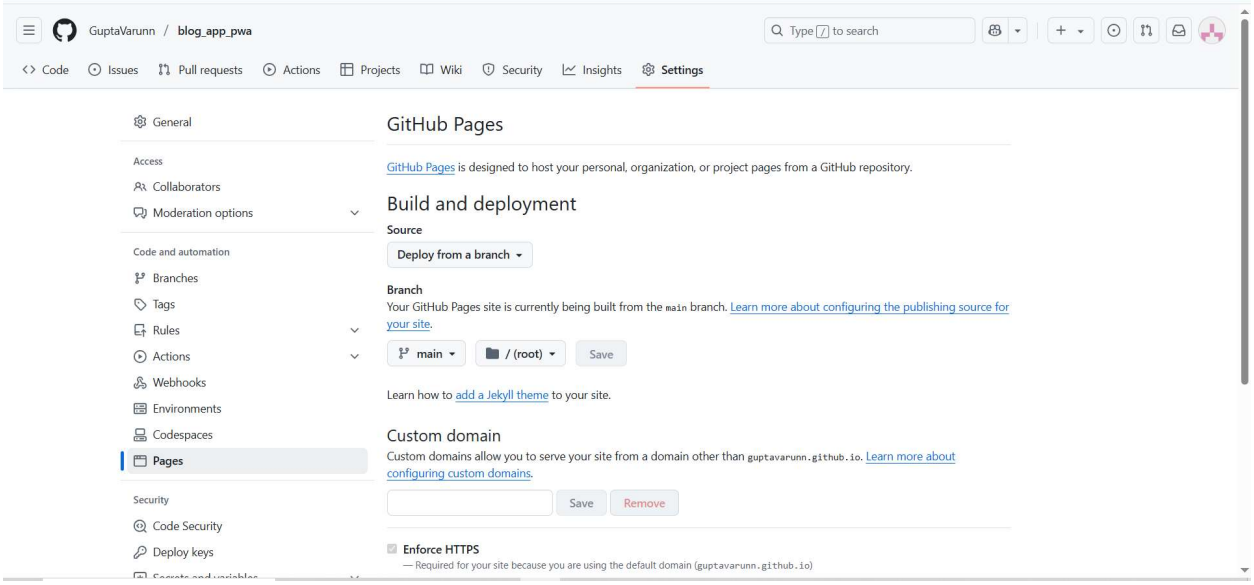
You can either:

- 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**

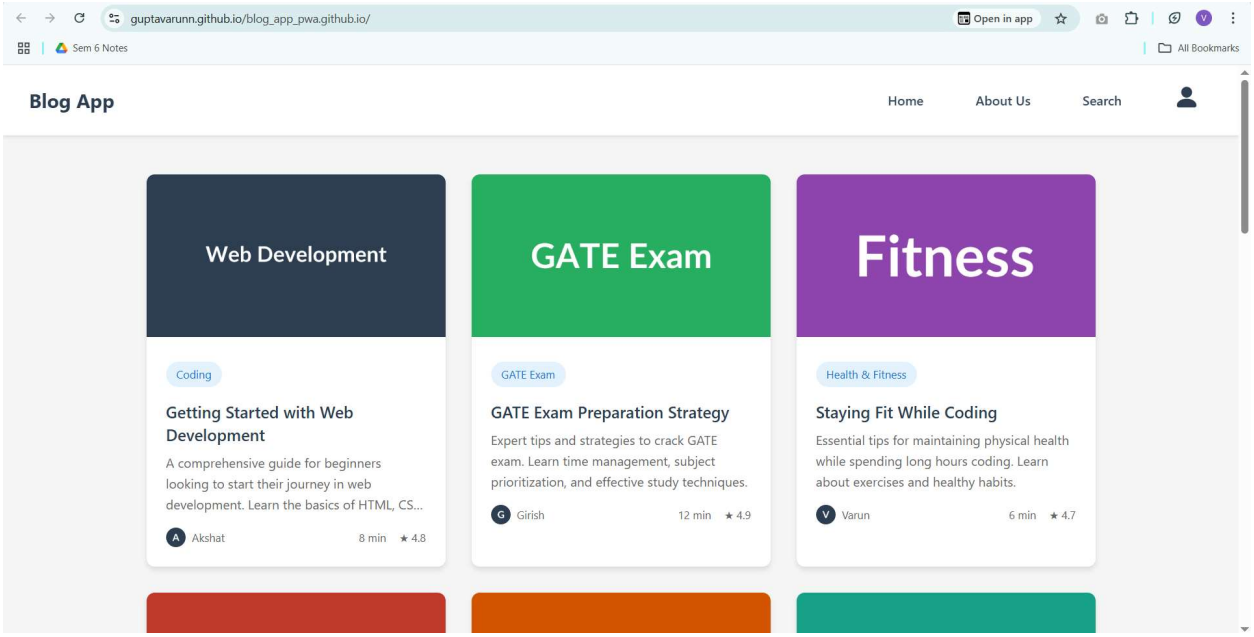


Step 4: Access Your Live Website

After enabling GitHub Pages:

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

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.

Lab 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**

---

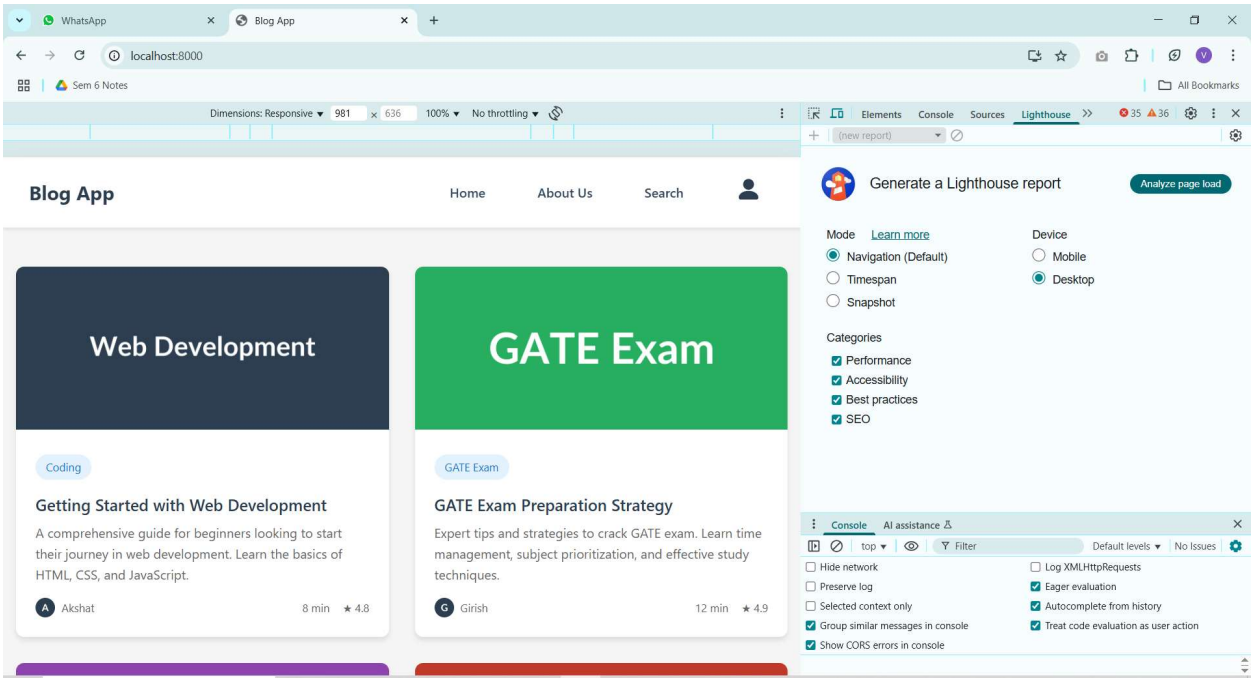
2. Open Chrome DevTools

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

---

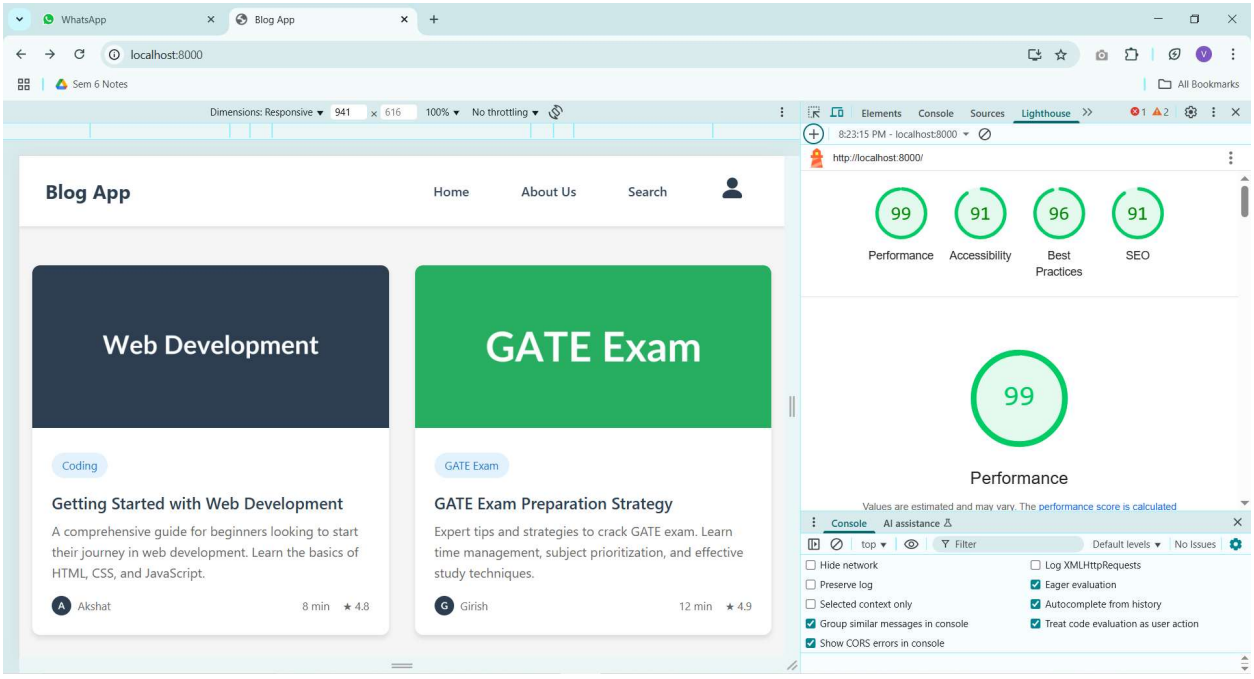
3. Configure Lighthouse Audit

- 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

VARUN GUPTA
D15B - 18
DATE:

AH 04

Q1. Explain key features & advantages of using flutter for mobile app development

Ans

A] Single codebase for multiple platforms. One codebase for both Android and iOS, reducing development effort.

i) Hot reload - instantly see changes in app without restarting making development faster

ii) Fast performance - use dart language and a compiled approach for smooth & high performance apps.

iii) Open source & strong community support - backed by Google and a large developer community, ensuring continous improvement

Advantages:

1) Faster development time: Hot reload & single codebase reduce development time.

2) Cost effective: since code runs of both Android and iOS, business save on development & maintainence.

3) Reduce performance issues: The app runs natively without relying on intermediate bridge like in react native.

B] Discuss how flutter framework differs from traditional approach and why it has gained popularity.

Ans:

Single codebase vs Separate codebase

Traditional Approach: Developers need to write separate code for Android & iOS

Flutter uses a single dart based codebase for platform reducing time & effort.

2. Rendering Engine vs Native UI

Traditional approach: relies on platform native UI component which can lead to inconsistency.

Flutter uses the skia rendering engine to draw environment from scratch ensuring consistent UI across devices.

Why Flutter gained popularity?

Ans.

i) Faster development with Hot Restart, can instantly UI changes without restarting app making development easier.

ii) Cross platform efficiency, business save time & resources by maintaining single codebase for multiple platform.

iii) Consistent UI accross devices since flutter does not rely on native components.

iv) Improved performance: No compilation and direct access to GPU rendering ensuring smooth animation.

Q2. Describe the concept of widget tree in flutter. Explain how widget composition is used to build complex UI.

Ans.

Widget Tree in Flutter

i) It is the fundamental structing UI of application.

ii) It is hierarchical arrangement of widget can be stateless or stately.

iii) Widget tree determine how UI is rendered and updated when changes occur.

Widget Composition in Flutter

i) It refers to building complex UI ~~of~~ by ~~widget composition~~ combining smaller reusable widget.

ii) Instead of creating large monolithic UI component, Flutter encourages breaking the UI into smaller widgets to be reused

eg.

```
class ProfileCard extends StatelessWidget {
    final String name;
    final String componentS;
  ProfileCard ({ required this.name, req. this.img_url });
  @override
  Widget build (Build context context) {
      return Card(
        child Column (
          children. I
          Image.network (image:_url')
          SizedBox (height: 10)
```

Benefits:

1) Reusability - Small widgets can be reused index parts of app.
2) Maintainability - Breaking UI into smaller widget makes it easier to debug & update.
3) Performance - Flutter efficiently rebuilds only the necessary parts of widget tree.

1] Firebase Authentication

Enables secure authentication using email/password phone no. & third party providers like Google, facebook & Apple.

2] Cloud Firestore - store and sync data in real time across device support structure data queries and offline access

eg.

```
Firebase Firestore instance collection ('user').add({
    'name': 'JohnDoe',
    'email': 'JohnDoe@gmail.com'
});
```

3] Realtime Database

A realtime JSON database that automatically updates data across devices.

eg. Database Refrence ref = Firebase.Database instance ref.
let ({"text": "Hello, Firebase"})

4] Firebase cloud messaging (FCM)

Enables push notification and messaging between users

ex. Fire messaging.instance subscribe ToTopic ("news")

5] Firebase Hosting

Deploys and serves webapp securely with automatic.

→ Data Synchronisation in Firebase

Firebase ensures real time data synchronisation across multiple devices and platforms using firestore and creating database.

1) Cloud firebase sync mechanism.

Use realtime listners to update UI instantly when data changes.

ex · Firebase · Firestore instanu. collections ("users"). snapshot()

for (var doc. snapshot. does)
print (doc [nami]);

→ Runtime Database sync. Mechanism.

uses persistent websocket connection you live. updates.

ex.

Database Reference refs = firebase. database. finstance.
ref ("Message").

ref on value listen ((event){
print (event. snapshot. value)
});

→ Offline Data sync.

Firestore caches data locally & sync changes when the developer online.

ex. Firebase firestore. instance. setting. setting.

→ Cloud function for automated updates.

Automates backend logic to trigger when data changes.

MPL LAB

VARUN GUPTA
D15B - 18

ASSIGNMENT 2

Q1. Define Progressive Web App (PWA) and explain its significance in modern web development. Discuss the key characteristics that differentiate PWAs from traditional Mobile Apps

Ans.

1) Progressive Web Apps (PWA) is a type of web application that provides app like experience while being accessible through a web browser.
2) PWAs leverage moder web technologies to deliver fast reliable web user interactions similar to native mobile apps.
3) They are designed to work on any devices and adapt seamlessly to different screen sizes.
4) By using technologies like Service Workers, Web App Manifest, HTTPs, PWAs enable feature such as offline access, push notification and background updates.
5) This makes them power alternative to traditional mobile apps.

Significance in Modern Web Development

1) Enhances performance & user engagement
2) Works offline and loads quickly even in low network
3) Eliminates need of app store downloads.

Key differences from Traditional Mobile Apps

1) No App store dependency - Runs directly on a browser.

2) Offline Functionality - uses cached data via Service Workers

3) Responsive Design - Adapts seamlessly to different screen sizes.

4) Lower Development cost - single codebase for web & mobile.

Q2. Define Responsive Web Design (RWD) and explain its importance in context of PWA. Compare & contrast responsive, fluid and adaptive web design approaches

Ans.

1) Responsive Web Design (RWD) is a web development approach that ensures website adjust seamlessly to different screen sizes, resolutions and orientations.
2) It uses fluid grids, flexible images and CSS media queries to dynamically change the layout based on the device
3) This ensures consistent and user friendly experience across desktops, tablets, and smartphones without requiring separate versions of website.
4) By making websites adaptable, RWD enables usability, accessibility and performance.

Importance in PWAs

PWAs are designed to work across multiple devices and platform making responsible web design a critical component. It provides

1) Seamless User Experience - Ensures smooth navigation to any screen size.
2) Cost Effective Development - Elimates the need of Separate mobile and desktop versions.
3) Better Accessibility and Engagement - Users can access PWAs on any devices without installation.
4) Improved Performance - optimized devices reduce load times and enhances usability.

Teacher's Sign.:

Feature	Responsive Design	Fluid Design	Adaptive Design
Layout Type	Flexible and adjusts based on screen size	Uses percentage base widths for smooth resizing	Predefining layout for specific screen sizes.
Flexibility	Highly flexible, adjusts dynamically	Fully scalable without breakpoints	Fixed layouts change at specific breakpoints
Best for	Multi-device compatibility (PWAs, modern website)	Smooth proportional scaling	Optimized experiences for specific devices.
Development complexity	Moderate	Simple	More complex due to multiple layouts.

Q3. Describe lifecycle of service workers, including registration, installation and activation phases.

Ans.

A service workers is a script that runs in the background, enabling features like offline caching, push notifications and background sync. in PWAs.
Its lifecycle consists of 3 main phases:

(1) Registration :

i. Service worker is registered in the browser using Javascript (navigator.serviceWorker.register() method)

ii. This ensures browser knows about service worker and can manage its lifecycle.

Teacher's Sign.:

How IndexDB is used in Service Workers.

(1) Storing Offline Data:

i) Service workers fetch data from network and store it in IndexedDB.

(ii) This enables app to display content even when offline.

(2) Efficient Data Retrieval:

i) Unlike Cache API, IndexedDB allows storing and querying structured data.

ii) It supports key-value pairs, indexes and transactions for efficient searching.

(3) Syncing Data with Server.

i) When user is offline, data (eg. form inputs, message) can be stored locally.

ii) Once connection restored, background sync updates server with stored data

(4) Improving Performance:

i) IndexedDB reduces unnecessary network requests by serving stored data

ii) This enhances the speed and responsiveness of PWAs.

eg. 1) Service Worker intercepts a fetch request.

2) It checks IndexedDB for stored data

3) If available, it serves the data, else it fetches from network and stores it.

Teacher's Sign.:

MAD LAB

VARUN GUPTA
D15B - 18

DATE:

PROJECT TOPIC SUMMARY

TOPIC NAME : News App

OBJECTIVES :

The objective of this project is to develop a user friendly News app using Flutter that provides real-time, categorized news updates. The app aims to deliver concise news summaries, improve user engagement through personalised content and seamless access to important news, anytime, anywhere.

KEY FEATURES :

1. Real-time News Feed : Displays latest news articles from various sources.
2. Text to Speech : Users will be able to listen to the news article, instead of reading it
3. Display themes : Night mode and other theme for personalisation as per user preferences.
4. Save later : Users will be able to bookmark specific articles to read them in future.
5. AI - News Summary Generator - Automatically shorten long articles into brief summaries using AI APIs