EXPERIMENT NO.2

Experiment No. 2: **To design Flutter UI by including common widgets.**	
ROLL NO	**2**
NAME	**Mohit Ailani**
CLASS	**D15-B**
SUBJECT	**MAD & PWA Lab**
LO-MAPPED	

Aim: To design Flutter UI by including common widgets.

Theory:
Flutter is an open-source UI toolkit developed by Google for building natively compiled applications across multiple platforms, including mobile, web, and desktop, from a single codebase. It utilizes the Dart programming language and offers a rich set of customizable widgets for creating beautiful, fast, and responsive user interfaces. Widgets in Flutter are building blocks of the UI, ranging from basic elements like buttons and text to complex layouts like lists and grids. These widgets are highly flexible, enabling developers to create immersive and dynamic user experiences with ease, making Flutter a powerful choice for modern app development.

Basic Widgets:

- **Appbar**: A fundamental Flutter widget representing a material design app bar at the top of the screen. It typically contains a title, leading and trailing icons, and actions. It's commonly used for navigation and displaying key information.

- **Scaffold**: A foundational widget providing layout structure for apps, offering features like app bars, drawers, bottom navigation, and floating action buttons, simplifying app development.

- **Column**: A layout widget arranging its children vertically from top to bottom, enabling flexible alignment and spacing along the vertical axis for building UIs.

- **Text**: A widget displaying text content with customizable properties like font size, color, alignment, and decoration, facilitating text rendering in Flutter apps.

- **Container**: A versatile widget serving as a visual element with customizable properties such as size, color, padding, margin, alignment, decoration, and child widget composition, offering flexibility in UI design and layout.

- **Row**: A layout widget arranging its children horizontally from left to right within the parent widget's constraints. It enables flexible alignment and spacing along the horizontal axis, allowing developers to create UIs with multiple widgets laid out side by side.

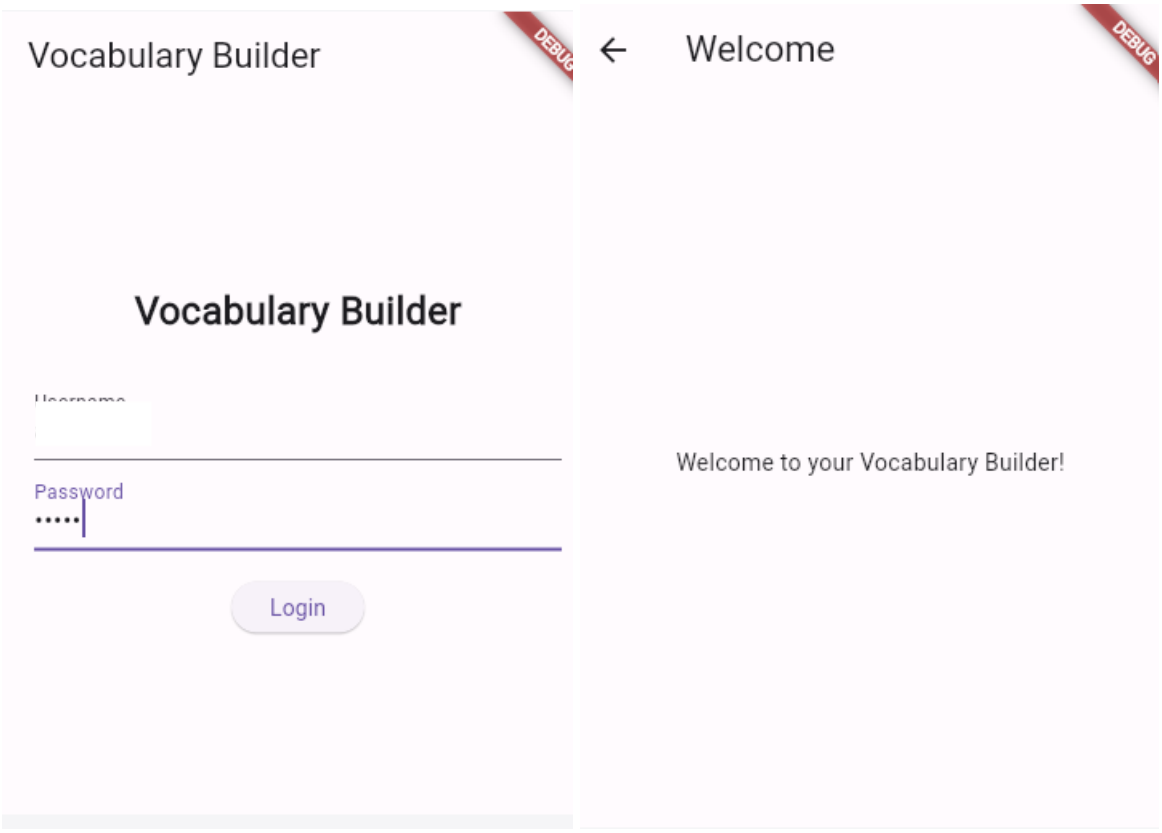
Code:

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

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

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

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

class _LoginPageState extends State<LoginPage> {
  final _formKey = GlobalKey<FormState>();

  String _username = "";
  String _password = "";

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Vocabulary Builder'),
      ),
      body: Form(
```

```
      key: _formKey,
      child: Padding(
        padding: const EdgeInsets.all(20.0),
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Center(
              child: Text(
                'Vocabulary Builder',
                style: TextStyle(fontSize: 24, fontWeight: FontWeight.bold),
              ),
            ),
            SizedBox(height: 20.0),
            TextFormField(
              decoration: InputDecoration(labelText: 'Username'),
              validator: (value) {
                if (value!.isEmpty) {
                  return 'Please enter your username';
                }
                return null;
              },
              onSaved: (value) => _username = value!,
            ),
            TextFormField(
              obscureText: true,
              decoration: InputDecoration(labelText: 'Password'),
              validator: (value) {
                if (value!.isEmpty) {
                  return 'Please enter your password';
                }
                return null;
              },
              onSaved: (value) => _password = value!,
            ),
            SizedBox(height: 20.0),
            ElevatedButton(
              onPressed: () {
                // Implement login logic here (replace with actual authentication)
                if (_formKey.currentState!.validate()) {
                  _formKey.currentState!.save();
                  Navigator.push(context, MaterialPageRoute(builder: (context) => WelcomePage()));
                }
              },
              child: Text('Login'),
            ),
          ],
        ),
      ),
    ),
  );
}
```

```
}

class WelcomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Welcome'),
      ),
      body: Center(
        child: Text('Welcome to your Vocabulary Builder!'),
      ),
    );
  }
}
```

Output:



Conclusion:

Flutter's common widgets, implementing AppBar, Scaffold, Column, and Row to design a comprehensive UI. This approach highlighted the versatility of Flutter in creating dynamic layouts while ensuring structural integrity and user interaction. Such implementation underscores Flutter's efficacy in developing intuitive and visually appealing applications.

EXPERIMENT NO.3

Experiment No 3 **To include icons, images, and fonts in Flutter app**	
ROLL NO	**2**
NAME	**Mohit Ailani**
CLASS	**D15-B**
SUBJECT	**MAD & PWA Lab**
LO-MAPPED	

Aim: To include icons, images, and fonts in Flutter app.

Theory:

Images in Flutter:

Images in Flutter are represented using the Image widget. They can be displayed from various sources such as assets, networks, memory, or files. Flutter supports popular image formats like PNG, JPEG, GIF, WebP, and BMP. You can customize the display of images using properties like fit, width, height, and more.

Icons in Flutter:

Icons in Flutter are vector graphics used to represent actions, categories, or entities in an app's user interface. Flutter includes a set of built-in Material Design icons and Cupertino icons (for iOS-style design). You can easily incorporate icons into your app using the Icon widget, specifying the desired icon from the available icon sets.

Fonts in Flutter:

Fonts in Flutter allow you to customize the typography and appearance of text in your app. Flutter supports both system fonts and custom fonts. Custom fonts can be declared in the pubspec.yaml file, specifying the font family name and the font files' paths. Once declared, you can use custom fonts by setting the fontFamily property in the TextStyle widget when styling text widgets.

Overall, images, icons, and fonts are fundamental elements in Flutter that contribute to creating visually appealing and engaging user interfaces in Flutter apps. They provide developers with the flexibility to customize the look and feel of their apps and enhance the user experience.

Code:

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

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

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

```
class MyHomePage extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Vocabulary Builder'),
      ),
      body: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Container(
              child: Image.asset('assets/images/vocab.png'),
              ),
            SizedBox(height: 20),
            // Displaying an icon from the assets folder
            Icon(
              Icons.favorite,
              size: 50,
              color: Colors.lightBlue,
            ),
            SizedBox(height: 20),
            // Using a custom font from the assets folder
            Text(
              'Importance of Vocabulary: Focusing on vocabulary is useful for developing knowledge and skills in multiple aspects of language and literacy. This includes helping with decoding (phonemic awareness and phonics), comprehension, and also fluency',
              style: TextStyle(
                fontFamily: 'ProtestRiot',
                fontSize: 24,
              ),
            ),
          ],
        ),
      ),
    );
  }
}
```

Output:

Conclusion: In conclusion, integrating icons, images, and fonts enhances Flutter apps, using packages like flutter_icons, and flutter_svg. These assets enrich the user experience, adding visual appeal and personality to the UI. With Flutter's asset management, developers can easily customize their app's appearance while maintaining performance. Overall, incorporating icons, images, and fonts elevates the design and usability of Flutter applications.

Name: mohit Ailani Roll No: 2 Div: D15B

MAD Lab Experiment No 5

Aim: To apply navigation, routing, and gestures in Flutter App

Theory:

Navigation

- Purpose: Navigation refers to the process of guiding users through the different screens (or "routes") within your app, allowing them to explore information and complete tasks. It establishes a clear flow and user experience.
- Techniques in Flutter:
    a. Navigator: Flutter's built-in Navigator class is the central component for managing navigation. It provides methods for pushing new routes onto the navigation stack (moving forward), popping routes from the stack (going back), and replacing the current route.
    b. Named Routes: Named routes associate a unique string identifier with each route, making navigation more readable and maintainable. You define these routes in the MaterialApp or WidgetsApp constructor's routes property.

Routing

- Purpose: Routing defines the logic behind how the app determines which route (screen) to display based on user actions or data. It establishes the mapping between events and destinations.
- Implementation in Flutter:
    a. Basic Navigation: For simple navigation, you can use the Navigator.push() and Navigator.pop() methods directly.
    b. Declarative Routing (Packages): For complex applications with deep linking or advanced navigation patterns, consider using third-party routing packages like go_router or fluro. These packages provide a more declarative approach to defining routes and handling transitions.

Gestures

- Purpose: Gestures are user interactions with the touch screen that control the app's behavior. They allow users to navigate, interact with UI elements, and manipulate data.
- Handling Gestures in Flutter:
    a. Gesture Detectors: Flutter provides various gesture detectors (like GestureDetector, TapGestureRecognizer, SwipeGestureRecognizer) to capture and interpret user gestures. These detectors trigger callbacks based on the type and timing of the gesture.

Code:

WelcomePage.dart:

```
import 'package:flutter/material.dart';
import 'package:flashcards_quiz/views/login_page.dart';

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

 @override
 Widget build(BuildContext context) {
  return Scaffold(
   backgroundColor: const Color.fromARGB(255, 215, 1, 11),
   body: Center(
    child: Column(
     mainAxisAlignment: MainAxisAlignment.center,
     children: [
      // Add your app logo image here
      Image.asset("assets/dictionary_img.png"),
      const SizedBox(height: 20),
      const Text(
       "Vocabulary Builder",
       style: TextStyle(
        fontSize: 30,
        fontWeight: FontWeight.bold,
        color: Colors.white,
       ),
      ),
      const SizedBox(height: 20),
      Padding(
       padding: const EdgeInsets.only(bottom: 30, right: 30),
       child: FloatingActionButton(
        onPressed: () => Navigator.push(
         context,
         MaterialPageRoute(builder: (context) => const LoginPage()),
        ),
        child: const Icon(Icons.arrow_forward),
       ),
      ),
```

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

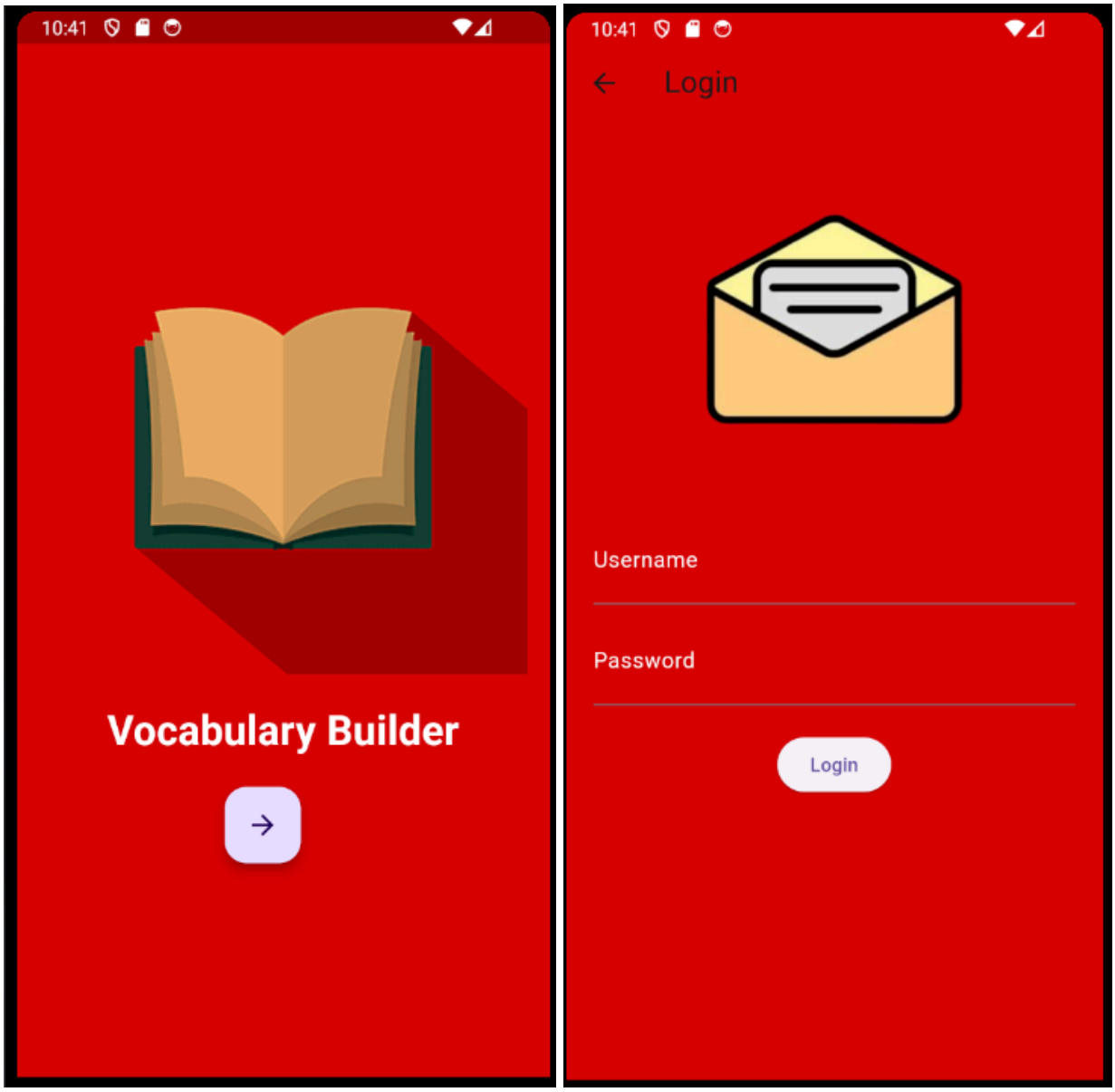
LoginPage.dart:

```
import 'package:flutter/material.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:flashcards_quiz/views/home_page.dart';

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

  @override
  State<LoginPage> createState() => _LoginPageState();
}

class _LoginPageState extends State<LoginPage> {
  final _usernameController = TextEditingController();
  final _passwordController = TextEditingController();
  final FirebaseAuth _auth = FirebaseAuth.instance;
  String _errorText = '';

  @override
  void dispose() {
    _usernameController.dispose();
    _passwordController.dispose();
    super.dispose();
  }

  Future<void> _signInWithEmailAndPassword() async {
    try {
      final UserCredential userCredential =
          await _auth.signInWithEmailAndPassword(
        email: _usernameController.text.trim(),
```

```
      password: _passwordController.text,
    );
    if (userCredential.user != null) {
      // Navigate to home page if authentication successful
      // ignore: use_build_context_synchronously
      Navigator.push(
        context,
        MaterialPageRoute(builder: (context) => const HomePage1()),
      );
    }
  } catch (e) {
    setState(() {
      _errorText =
          'Invalid email or password'; // Set error message for invalid credentials
    });
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text("Login"),
      backgroundColor: const Color.fromARGB(255, 215, 1, 11),
    ),
    backgroundColor: const Color.fromARGB(255, 215, 1, 11),
    body: SingleChildScrollView(
      // Make content scrollable
      padding: const EdgeInsets.all(20.0),
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          // Add your image widget here
          Image.asset(
            "assets/login_imgg.png",
            width: 250,
            height: 250,
          ),
```

```
          const SizedBox(height: 20),
          TextField(
            controller: _usernameController,
            decoration: const InputDecoration(
              labelText: "Username",
              labelStyle: TextStyle(
                color: Colors.white, // Set text color
              ),
            ),
          ),
          const SizedBox(height: 10),
          TextField(
            controller: _passwordController,
            obscureText: true,
            decoration: InputDecoration(
              labelText: "Password",
              labelStyle: const TextStyle(
                color: Colors.white, // Set text color
              ),
              errorText: _errorText.isNotEmpty ? _errorText : null,
              errorStyle: const TextStyle(
                  color: Colors.white), // Set error text color
            ),
          ),
          const SizedBox(height: 20),
          ElevatedButton(
            onPressed:
                _signInWithEmailAndPassword, // Call method for authentication
            child: const Text("Login"),
          ),
        ],
      ),
    ),
  );
 }
}
```

Output:



Conclusion: The integration of navigation, routing, and gestures in Flutter app development significantly enhances user experience and interaction. Leveraging Flutter's robust navigation system, hierarchical routing, and diverse gesture recognizers, developers can create intuitive, seamless, and engaging applications. By prioritizing user-centric design and functionality.

Name: Mohit Ailani Roll No: 2 Div: D15B

MAD Lab Experiment No 6

Aim: To connect Flutter UI with Firebase.

Theory: Firebase is a Backend-as-a-Service (BaaS) platform from Google that provides a suite of tools to simplify mobile app development. Integrating Firebase with Flutter allows you to leverage these tools within your Flutter application for functionalities like authentication, database management, storage, analytics, and more.
Here's a breakdown of key concepts related to Firebase in Flutter:
1. Firebase Services:

- Authentication: Provides features for user signup, login, password reset, social logins (Google, Facebook, etc.), and user management.
- Firestore: A NoSQL cloud database for storing and retrieving structured data in a flexible and scalable manner.
- Storage: Cloud storage for images, videos, and other files, accessible from your app.
- Cloud Functions: Serverless functions that you can write and deploy to handle backend logic without managing servers.
- Real-time Database: A database that synchronizes data across devices in real-time, ideal for collaborative apps or live updates.
- Analytics: Tracks user behavior and app usage to gather insights into user engagement and app performance.
- Remote Config: Allows you to dynamically change app configurations without app updates, suitable for A/B testing or feature flags.
- Crashlytics: Provides crash reporting and analysis tools to help identify and fix bugs in your app.

2. Integration with Flutter:

- FlutterFire Plugins: Firebase provides official Flutter plugins (e.g., firebase_auth, cloud_firestore) for each service that simplify integration with your Flutter app.
- Setting Up Firebase Project: You'll need to create a Firebase project and configure it in your Flutter code using the provided web interface and configuration files.
- Adding Dependencies: Include the necessary FlutterFire plugins for the Firebase services you want to use in your pubspec.yaml file.

3. Using Firebase Services in Your App:

- Authentication Example:
    1. Initialize Firebase Auth using FirebaseAuth.instance.

2. Implement user signup or login logic using methods like createUserWithEmailAndPassword or signInWithEmailAndPassword.
3. Securely store user credentials or tokens using secure storage mechanisms.

- Firestore Example:
    1. Initialize Firestore using FirebaseFirestore.instance.
    2. Access collections and documents within your Firestore database.
    3. Use methods like add, get, set, and update to manage data.
- Storage Example:
    1. Initialize Cloud Storage using FirebaseStorage.instance.
    2. Upload or download files using methods like putFile and downloadURL.

4. Benefits of Using Firebase in Flutter:

- Reduced Development Time: Firebase provides pre-built services, saving you time on writing backend code.
- Scalability: Firebase services are automatically scalable, handling increased app usage without infrastructure management.
- Offline Support (Firestore and Realtime Database): Certain Firebase services offer offline capabilities for a seamless user experience.
- Realtime Features: Firebase Realtime Database enables real-time data synchronization across devices.
- Analytics and Monitoring: Firebase provides tools to track app usage and identify issues.

Steps to Set Up Firebase:

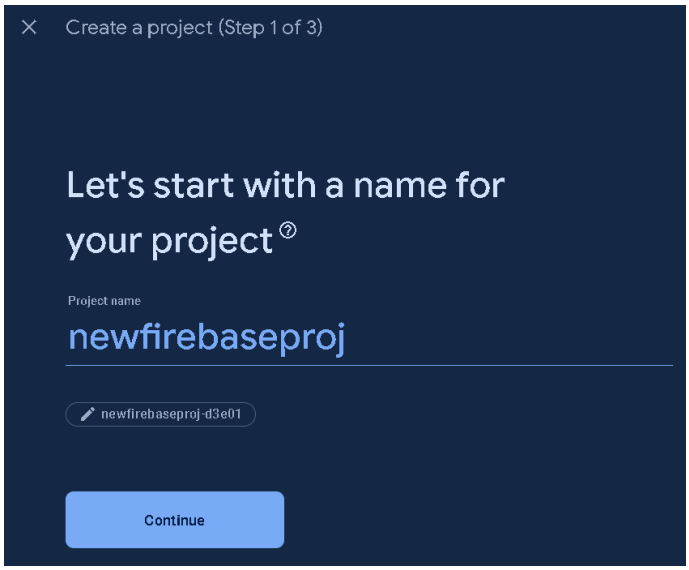
Prerequisites

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

Creating a New Flutter Project:

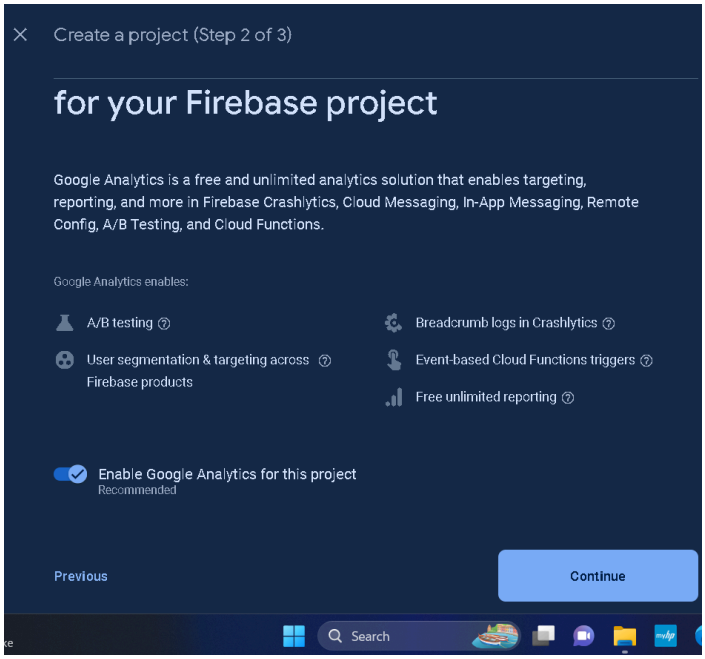
This tutorial will require the creation of an example Flutter app.
Once you have your environment set up for Flutter, you can run “flutter create <appname>” command.

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.



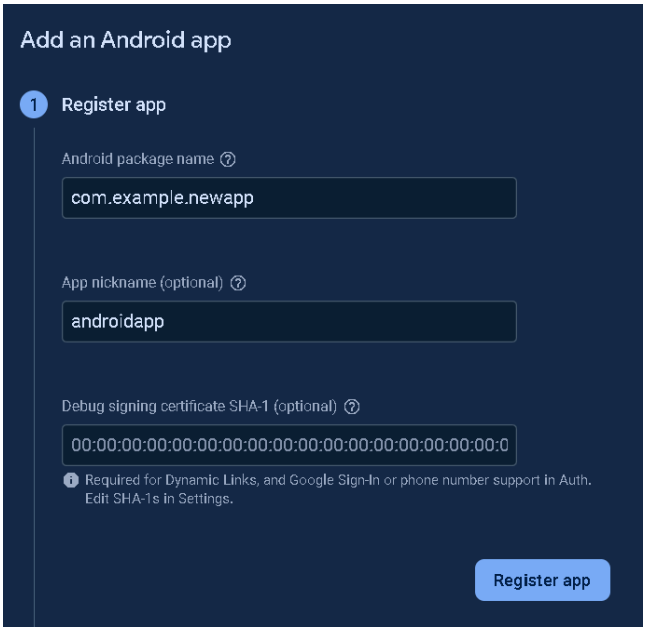
If you choose to use Google Analytics, you will need to review and accept the terms and conditions prior to project creation.

After pressing Continue, your project will be created and resources will be provisioned. You will then be directed to the dashboard for the new project.

Adding Android support:

1. 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.

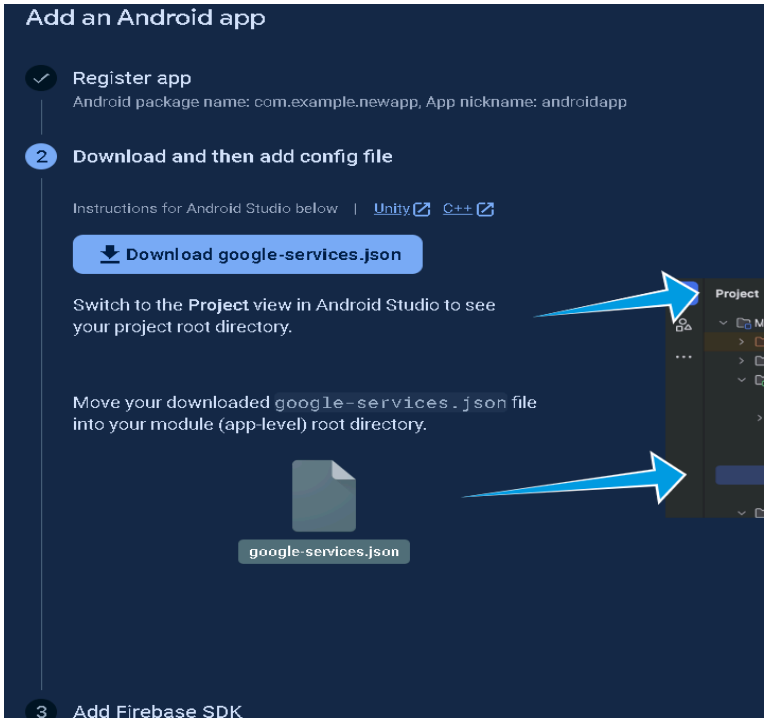
The structure consists of at least two segments. A common pattern is to use a domain name, a company name, and the application name: com.example.flutterfirebaseexample

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.Select Register app to continue.

2. 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.

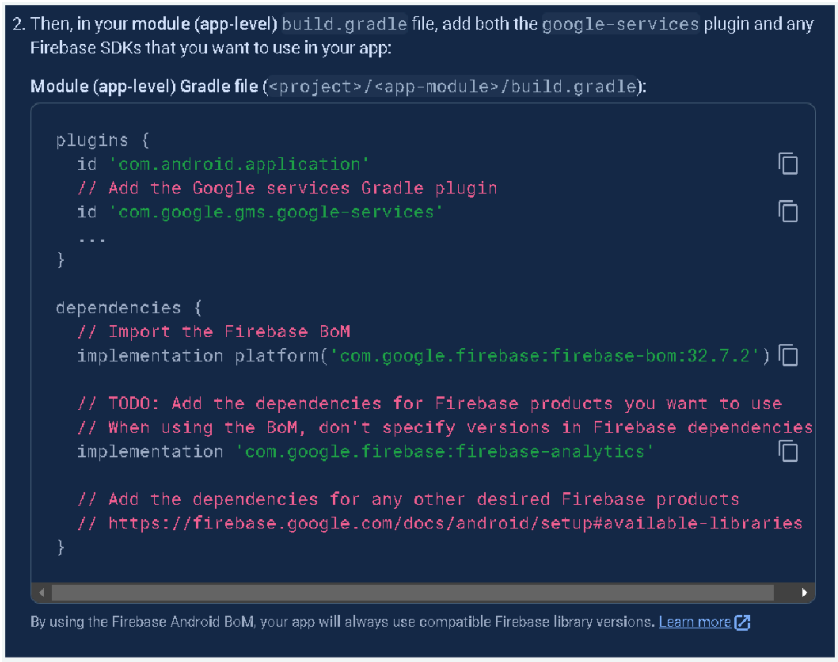
Select Download google-services.json from this page:



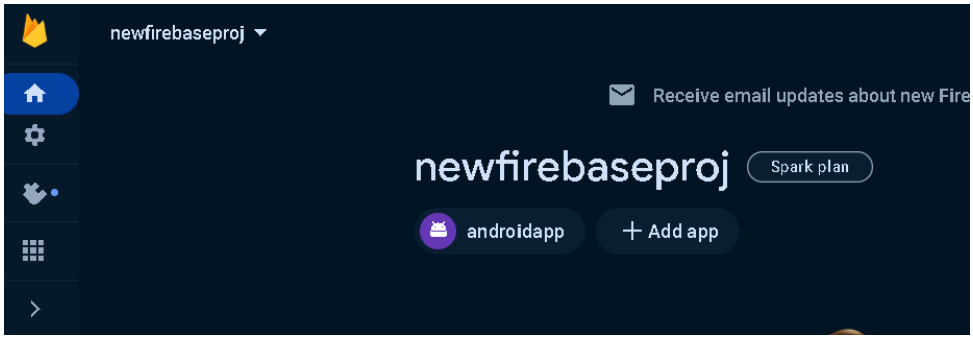
Next, move the google-services.json file to the android/app directory within the Flutter project.

3. Adding the Firebase SDK

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



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.



Using Firebase Authentication Functionality

Get Started with Firebase Authentication on Flutter

Send feedback

Connect your app to Firebase

Install and initialize the Firebase SDKs for Flutter if you haven't already done so.

Add Firebase Authentication to your app

1. From the root of your Flutter project, run the following command to install the plugin:

```
flutter pub add firebase_auth
```

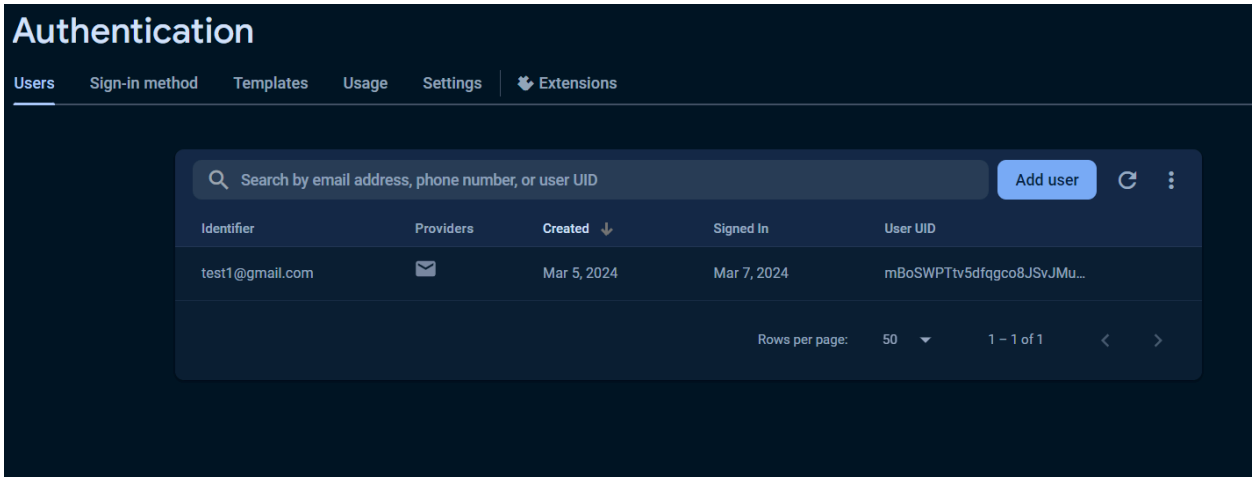
2. Once complete, rebuild your Flutter application:

```
flutter run
```

3. Import the plugin in your Dart code:

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

Creating a test case for authentication purpose:




```
dependencies:
  flutter:
    sdk: flutter

  firebase_core: ^2.26.0
  firebase_auth: ^4.17.7
```

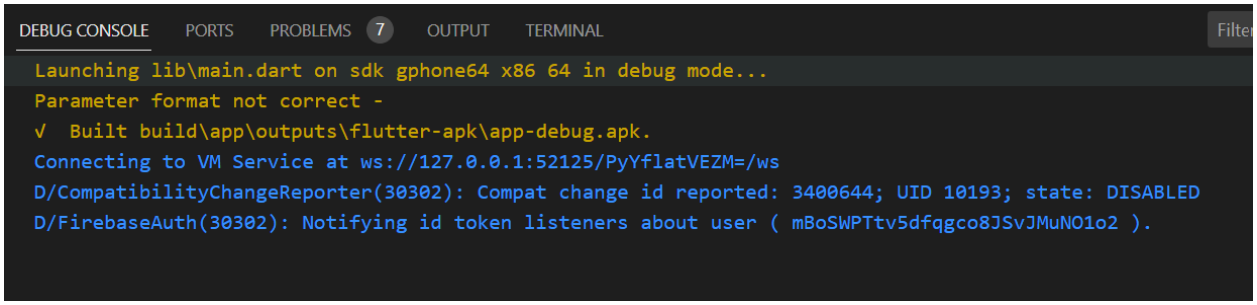
Code:

```
import 'package:flutter/material.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:flashcards_quiz/views/home_page.dart';

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

  @override
  State<LoginPage> createState() => _LoginPageState();
}

class _LoginPageState extends State<LoginPage> {
  final _usernameController = TextEditingController();
  final _passwordController = TextEditingController();
  final FirebaseAuth _auth = FirebaseAuth.instance;
  String _errorText = '';

  @override
  void dispose() {
    _usernameController.dispose();
    _passwordController.dispose();
    super.dispose();
  }

  Future<void> _signInWithEmailAndPassword() async {
    try {
      final UserCredential userCredential =
          await _auth.signInWithEmailAndPassword(
        email: _usernameController.text.trim(),
        password: _passwordController.text,
      );
      if (userCredential.user != null) {
        // Navigate to home page if authentication successful
        // ignore: use_build_context_synchronously
        Navigator.push(
          context,
          MaterialPageRoute(builder: (context) => const HomePage1()),
```

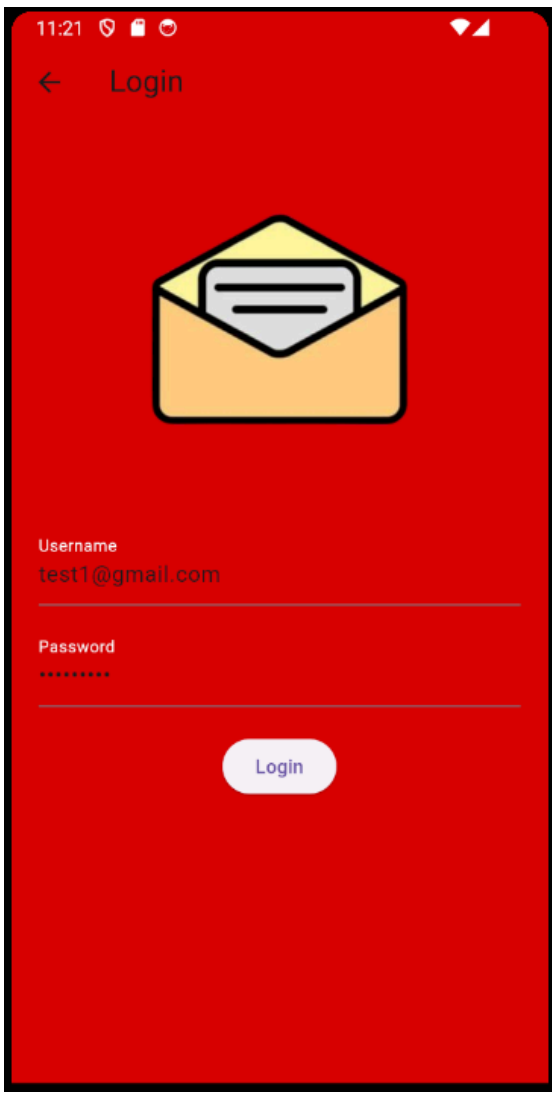
```
      );
    }
  } catch (e) {
    setState(() {
      _errorText =
          'Invalid email or password'; // Set error message for invalid credentials
    });
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text("Login"),
      backgroundColor: const Color.fromARGB(255, 215, 1, 11),
    ),
    backgroundColor: const Color.fromARGB(255, 215, 1, 11),
    body: SingleChildScrollView(
      // Make content scrollable
      padding: const EdgeInsets.all(20.0),
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          // Add your image widget here
          Image.asset(
            "assets/login_imgg.png",
            width: 250,
            height: 250,
          ),
          const SizedBox(height: 20),
          TextField(
            controller: _usernameController,
            decoration: const InputDecoration(
              labelText: "Username",
              labelStyle: TextStyle(
                color: Colors.white, // Set text color
              ),
```

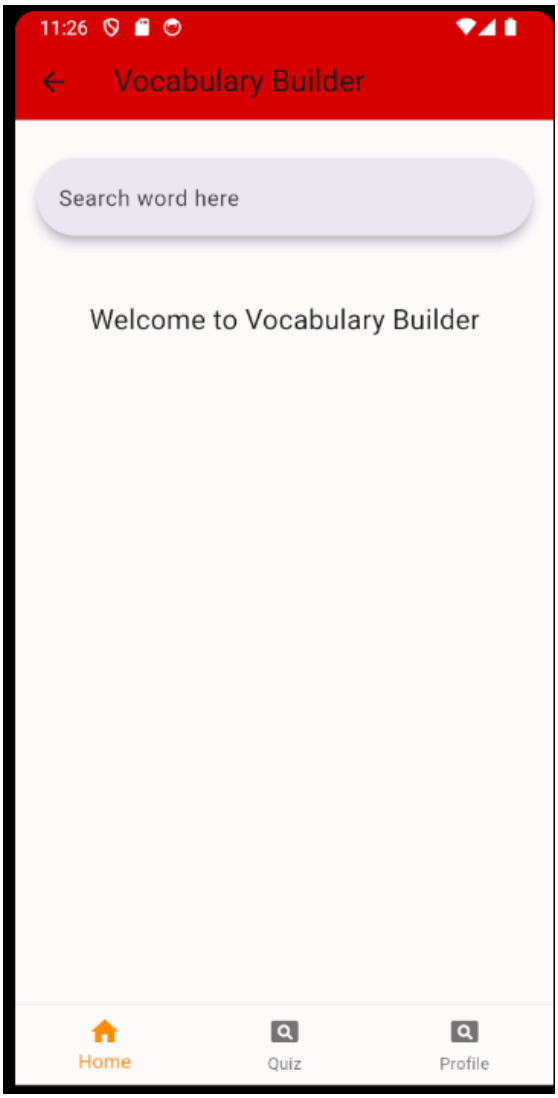
```
          ),
        ),
        const SizedBox(height: 10),
        TextField(
          controller: _passwordController,
          obscureText: true,
          decoration: InputDecoration(
            labelText: "Password",
            labelStyle: const TextStyle(
              color: Colors.white, // Set text color
            ),
            errorText: _errorText.isNotEmpty ? _errorText : null,
            errorStyle: const TextStyle(
                color: Colors.white), // Set error text color
          ),
        ),
        const SizedBox(height: 20),
        ElevatedButton(
          onPressed:
              _signInWithEmailAndPassword, // Call method for authentication
          child: const Text("Login"),
        ),
      ],
    ),
  ),
 );
 }
}
```

After running the code check whether the firebase connection is established with the project or not:



Output:





Conclusion : From this experiment, first of all we studied how to setup firebase. Then we created a firebase project, added multiple dependencies and packages to our flutter project so as to integrate our flutter project with firebase. Next step we authenticate the input fields from our app like email and password using Firebase and store it.

Experiment 1

Experiment No 1 **1: Installation and Configuration of Flutter Environment.**	
ROLL NO	**2**
NAME	**mohit ailani**
CLASS	**D15-B**
SUBJECT	**MAD & PWA Lab**
LO-MAPPED	

Experiment 1

Aim: Installation and Configuration of Flutter Environment.

Theory: Flutter is an open-source UI software development toolkit created by Google. It is used for building natively compiled applications for mobile, web, and desktop from a single codebase. Flutter enables developers to use a single programming language, Dart, to create applications for multiple platforms, streamlining the development process and reducing the need for separate codebases for iOS and Android.

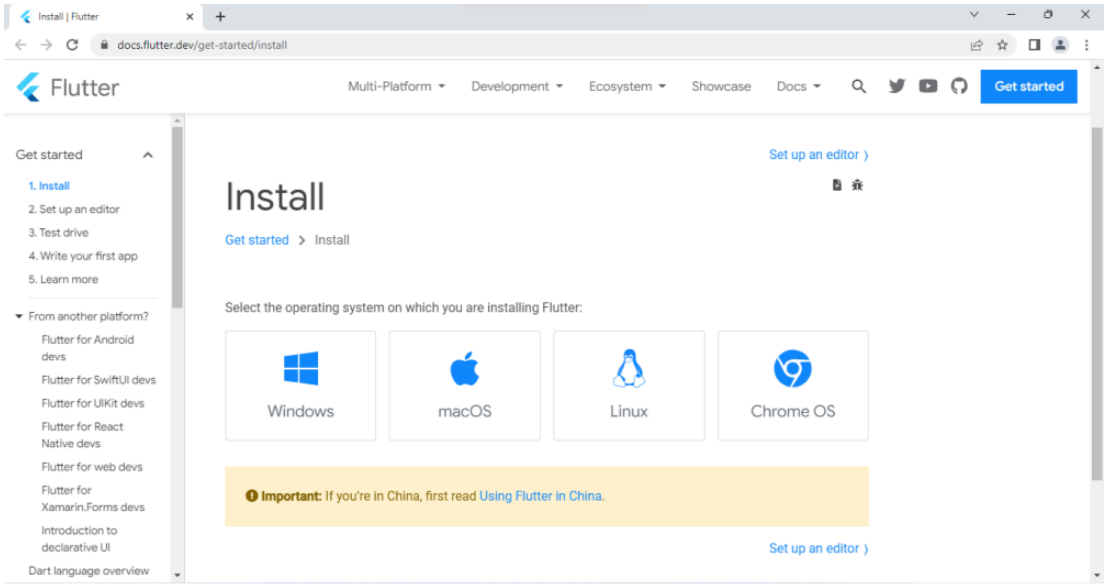
Key features of Flutter include:

1. **Hot Reload**: Developers can make changes to the code and see the results in real-time without restarting the application. This feature accelerates the development process and makes it easier to experiment with different UI elements.
2. **Widgets:** Flutter applications are built using a reactive framework composed of widgets. Widgets are reusable building blocks that help create the user interface. Flutter provides a rich set of pre-designed widgets for common UI elements and allows developers to create custom widgets.
3. **Single Codebase:** With Flutter, developers can write a single codebase that can be deployed on multiple platforms, including iOS, Android, web, and desktop. This can save time and resources compared to developing separate codebases for each platform.
4. **High Performance:** Flutter compiles to native ARM code, providing high performance that is comparable to natively developed applications. This is achieved through the use of the Skia graphics engine.
5. **Expressive UI:** Flutter allows developers to create highly customized and visually appealing user interfaces. The framework provides extensive support for animations and allows for pixel-perfect designs.
6. **Community and Ecosystem:** Flutter has a growing and active community of developers, which means there is a wealth of resources, plugins, and packages available to enhance the development process.

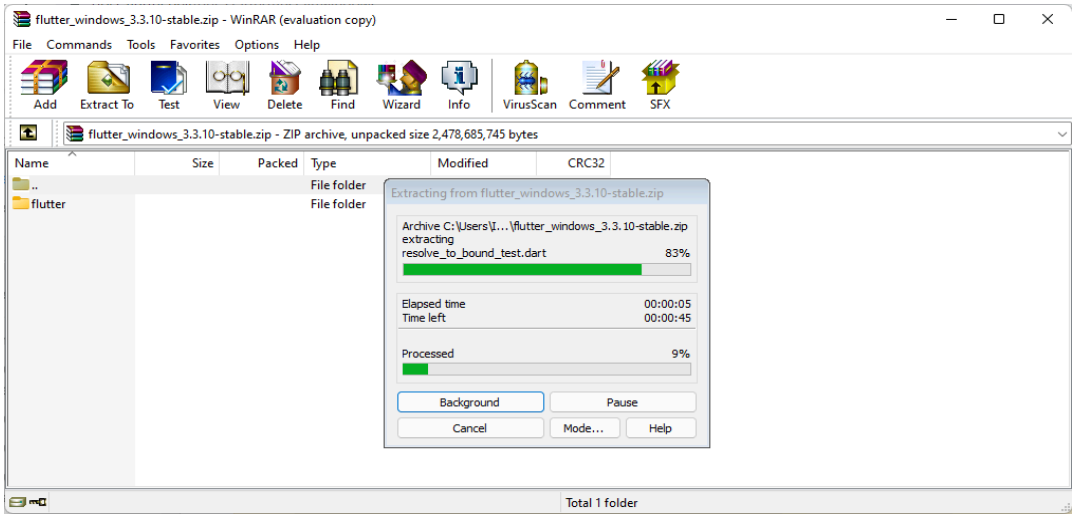
Flutter has gained popularity for its efficiency, flexibility, and ability to create visually appealing and high-performance applications across different platforms.

Flutter Installation:

Step 1: Install the Flutter SDK Download the Flutter Software Development Kit from the official website for Windows.



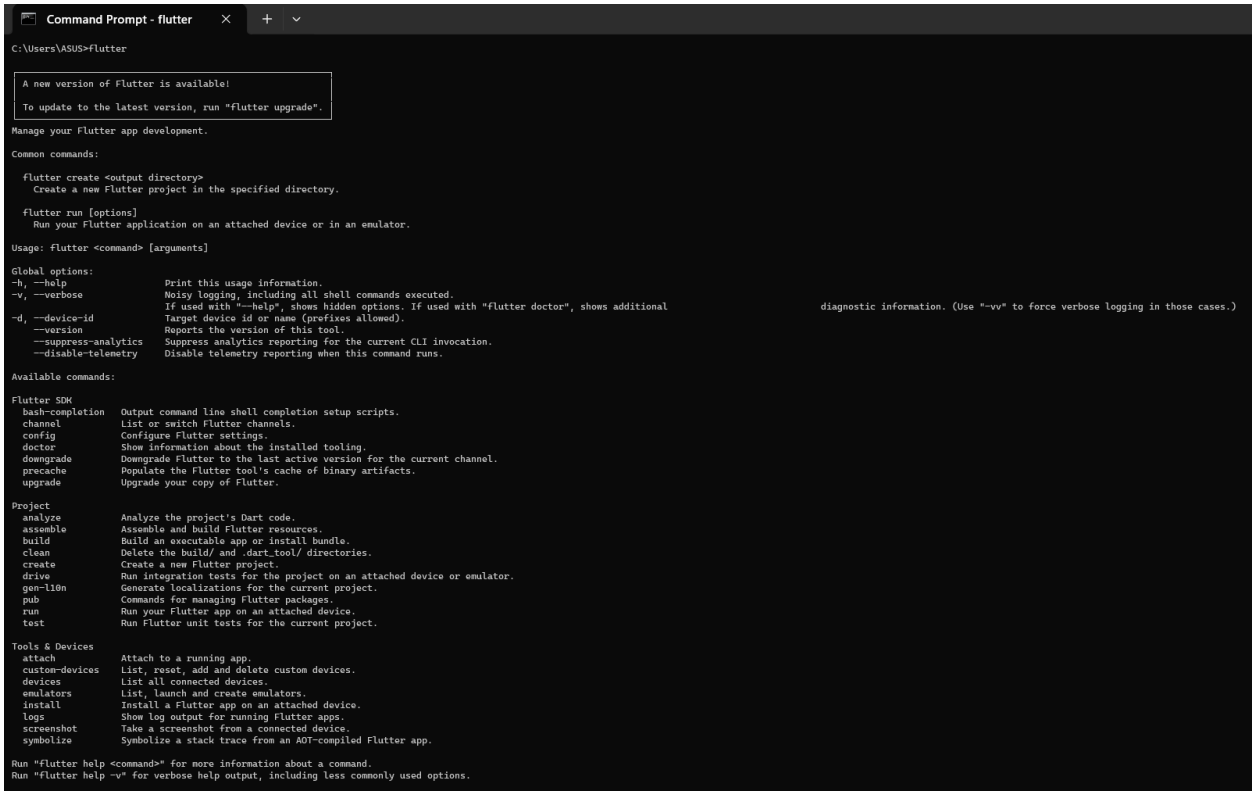
Step 2: Once the download is complete extract the zip file and place it in the desired folder.



Step 3: 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: Go to THIS PC -> Properties -> Advanced system settings -> Environment variables.

Step 4: Select the Path -> click on Edit and add the path to the Flutter bin folder.

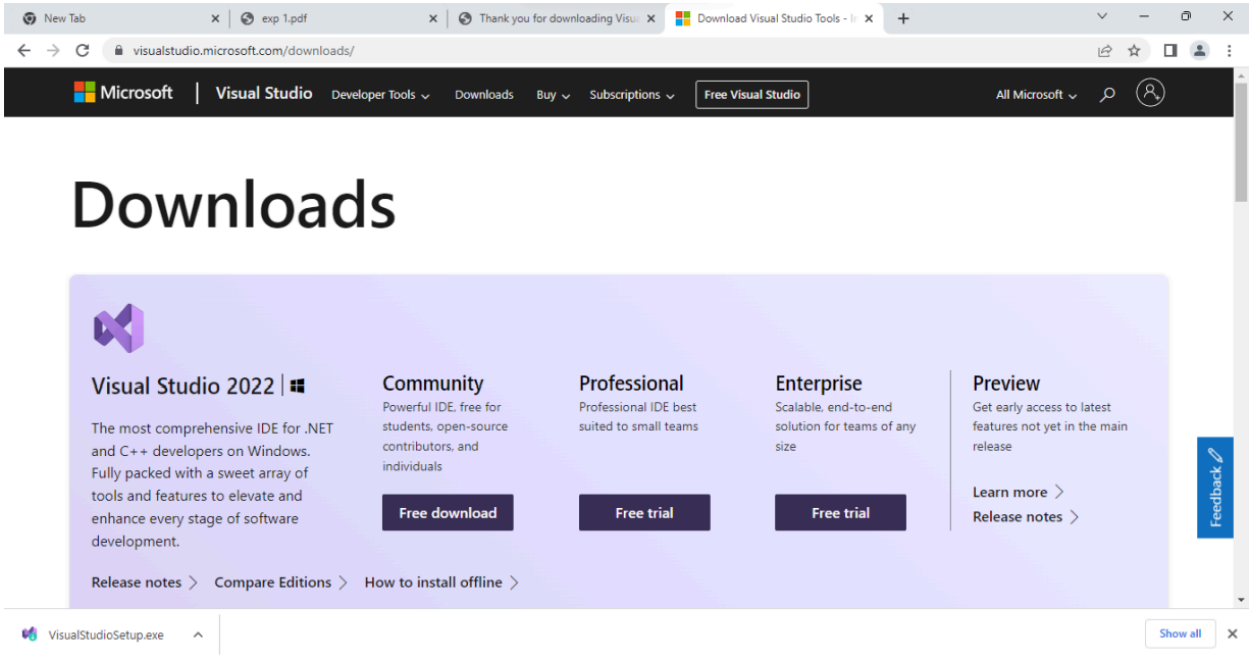
Step 5: Run the flutter command in the Command Prompt.

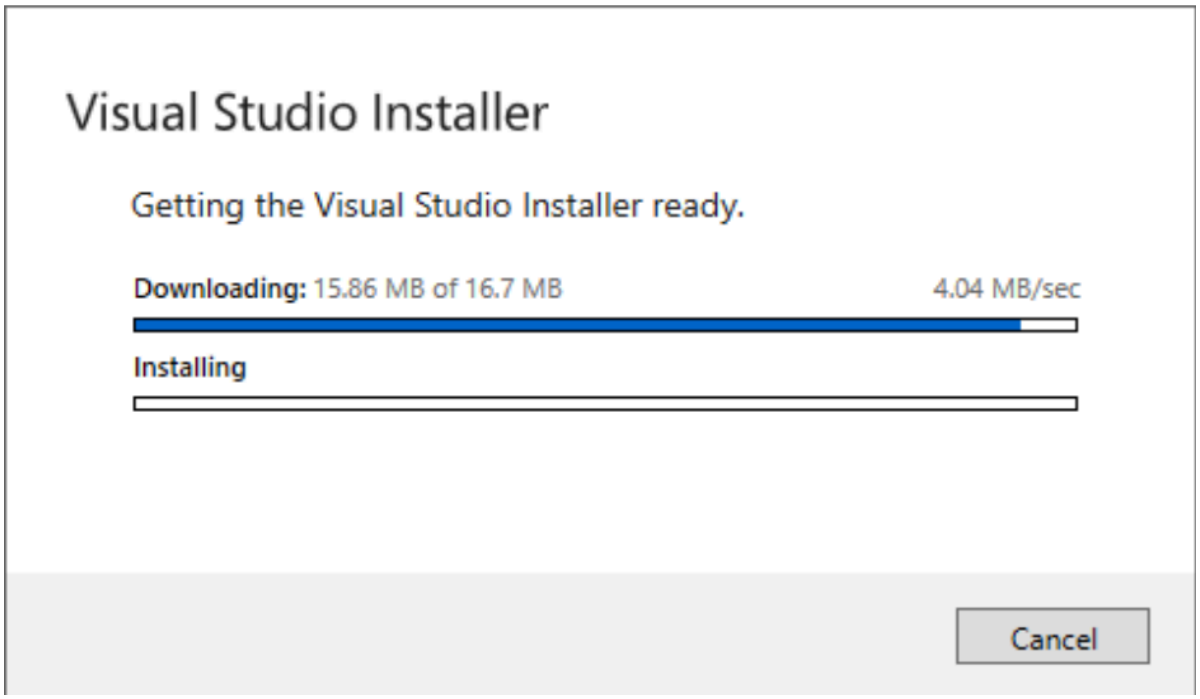


Also, run the flutter doctor command. This command checks for all the requirements of Flutter app development and displays a report of the status of your Flutter installation.

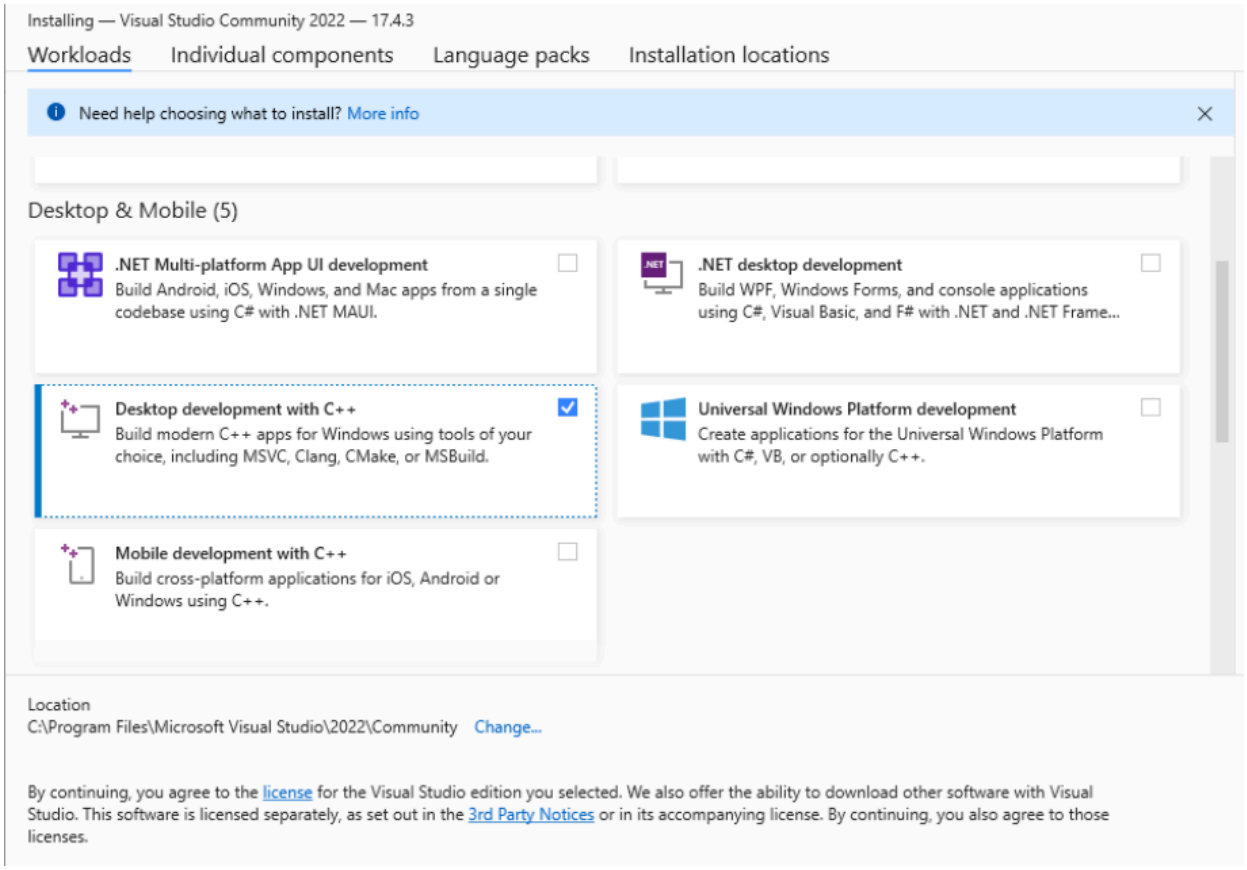
Step 6: Now we will resolve all the issues and install and add the missing tools required for Flutter app development.

Step 6.1: Download and install Visual Studio.



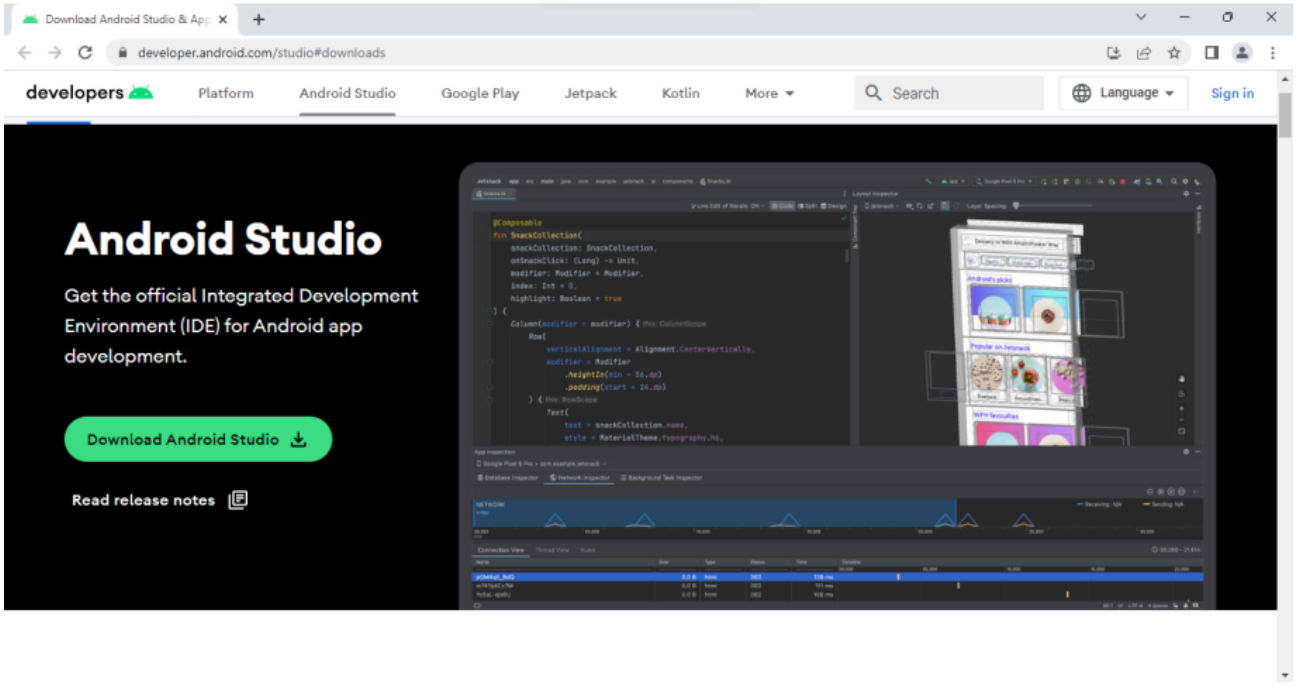


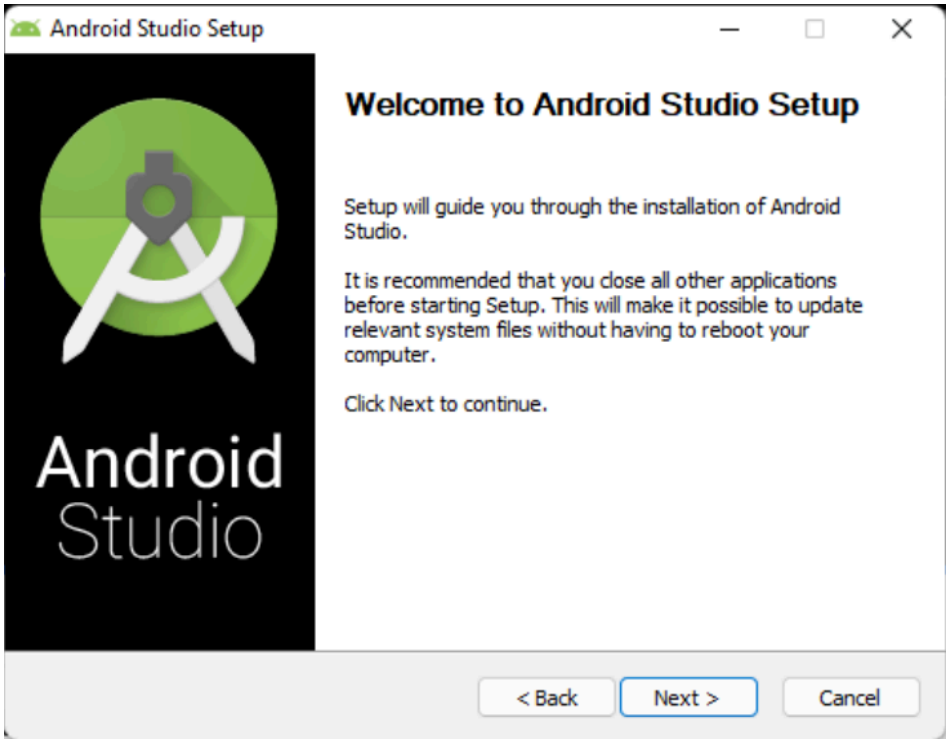
While Installing Visual Studio, make sure to select Desktop development with C++.

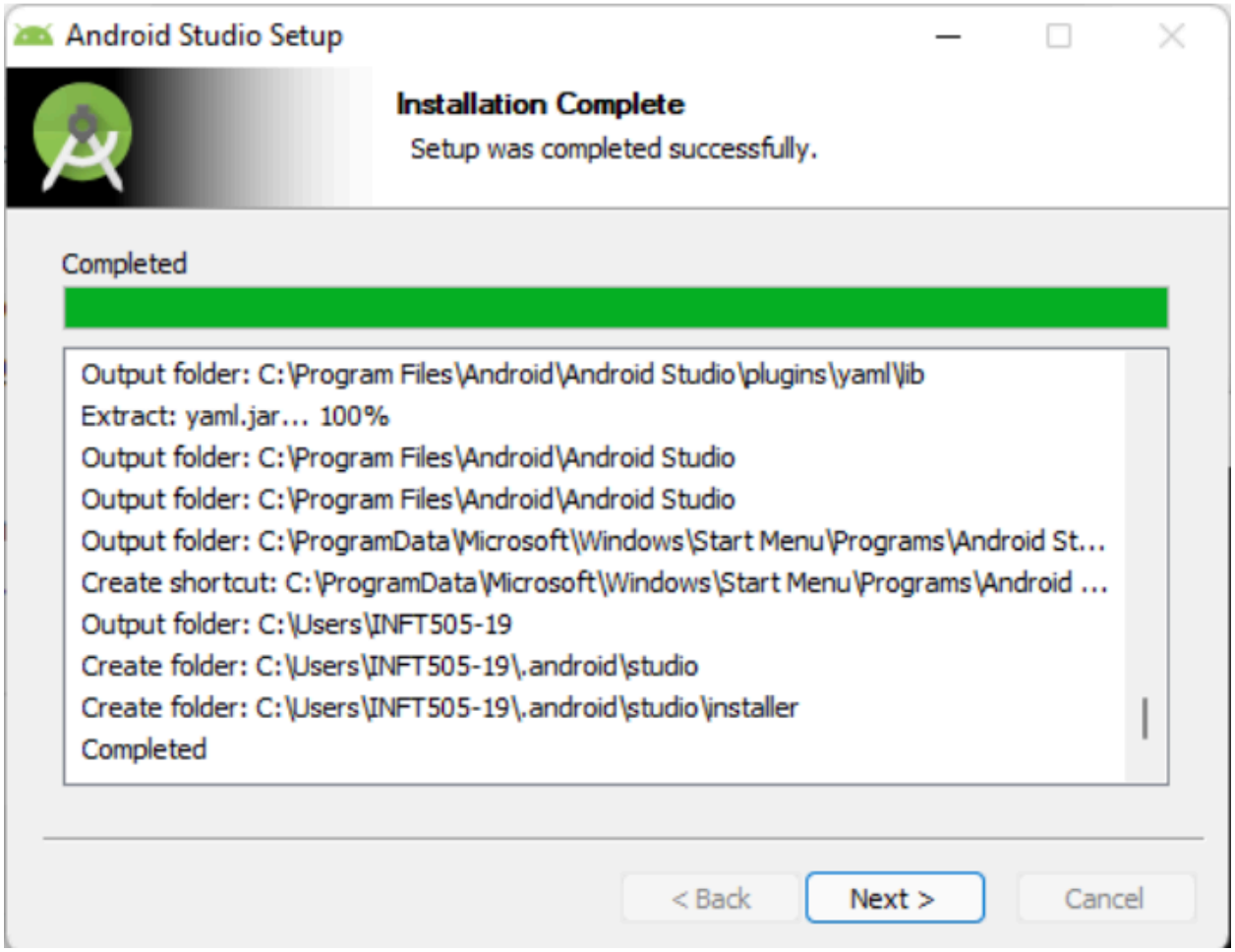


We can see that the Visual Studio issue has been solved.

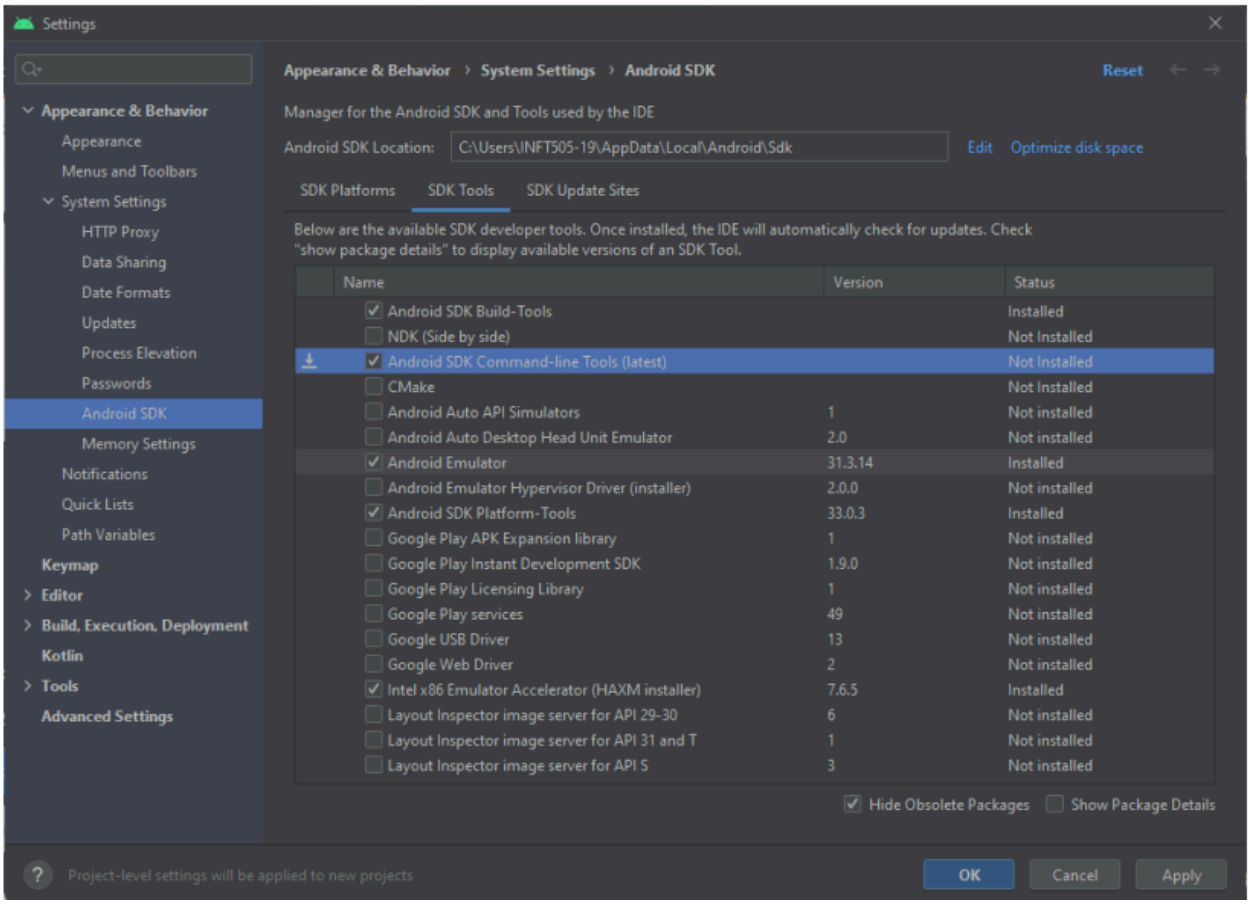
Step 6.2: Install the Android SDK. If the flutter doctor command does not find the Android SDK tool in your system, then you need first to install the Android Studio IDE.


Download Android Studio & App
developer.android.com/studio#downloads
developers
Platform
Android Studio
Google Play
Jetpack
Kotlin
More
Search
Language
Sign in
Android Studio
Get the official Integrated Development Environment (IDE) for Android app development.
Download Android Studio
Read release notes


Android Studio Setup
Welcome to Android Studio Setup
Setup will guide you through the installation of Android Studio.
It is recommended that you close all other applications before starting Setup. This will make it possible to update relevant system files without having to reboot your computer.
Click Next to continue.
Android Studio
< Back
Next >
Cancel

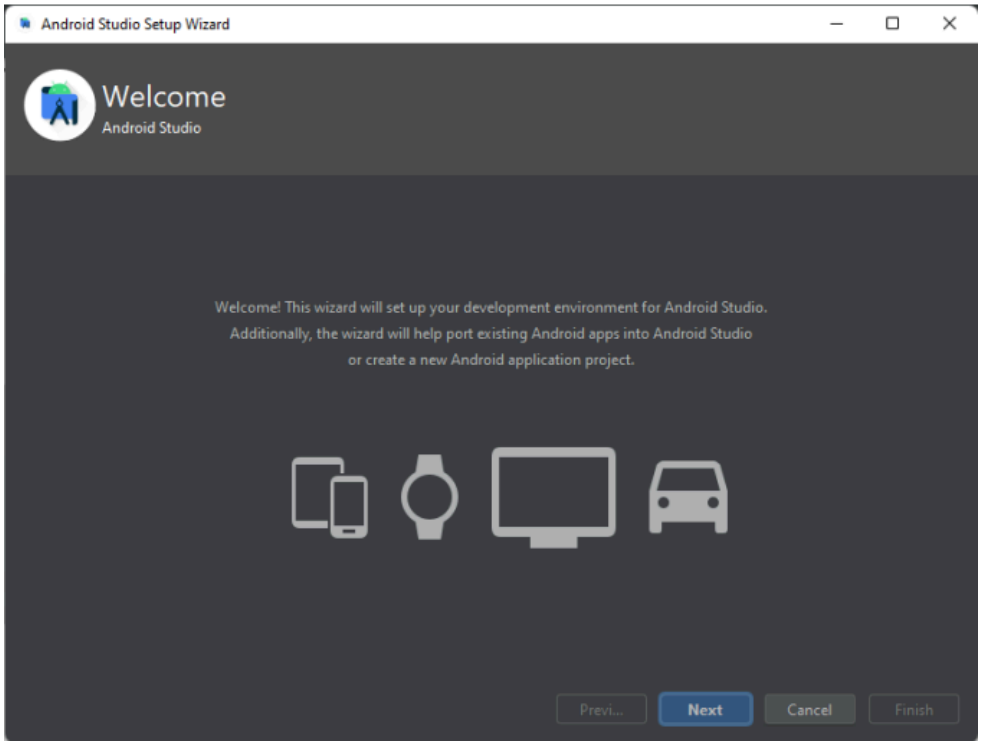


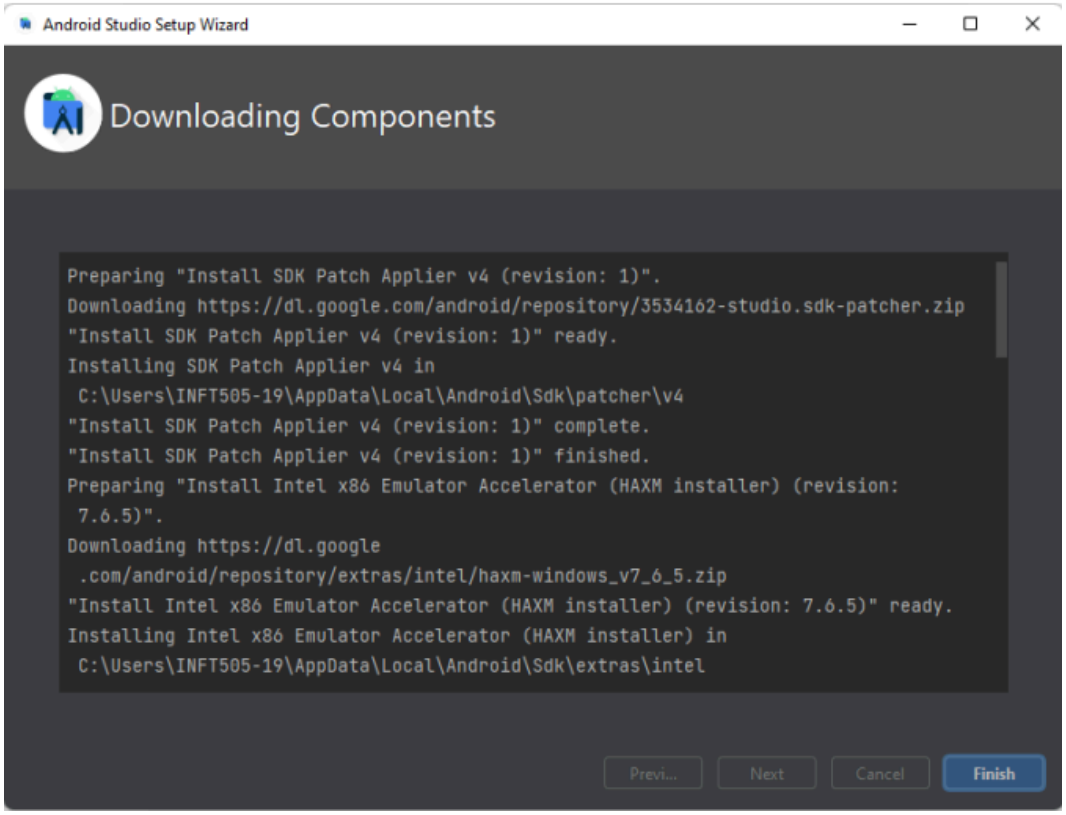
Once the android Studio is installed, Download the Android SDK Command-Line Tools present in the Android SDK section in Android Studio.



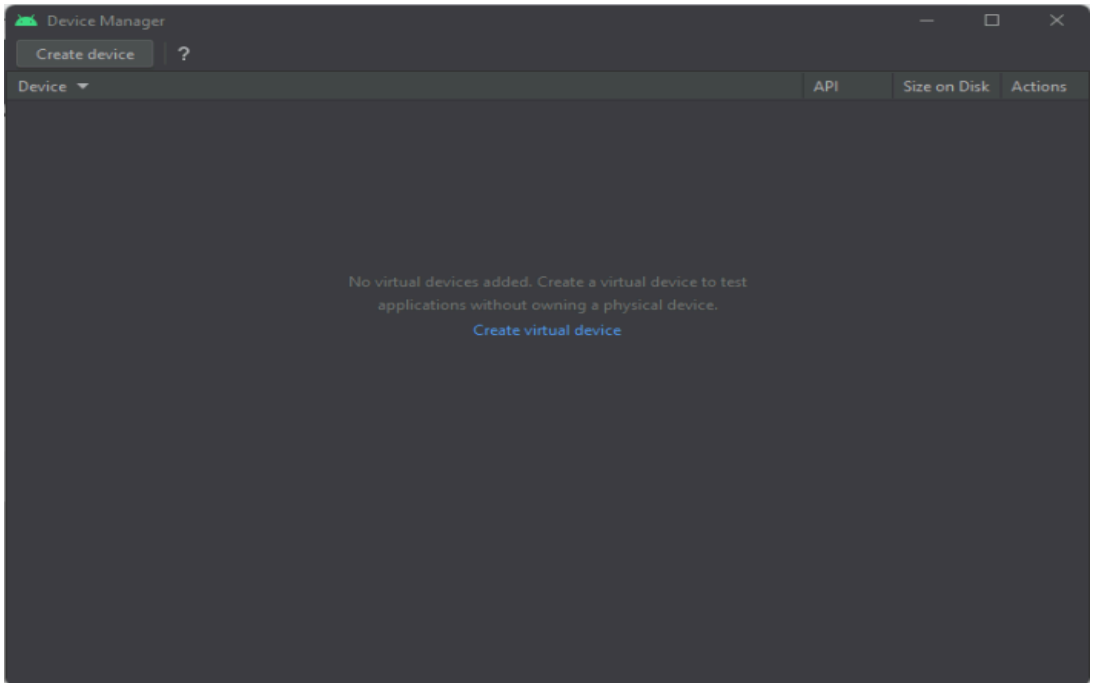
Run the flutter doctor command and run flutter doctor –android licenses to accept all the required licenses.

Step 7: Next, you need to set up an Android emulator. It is responsible for running and testing the Flutter application.

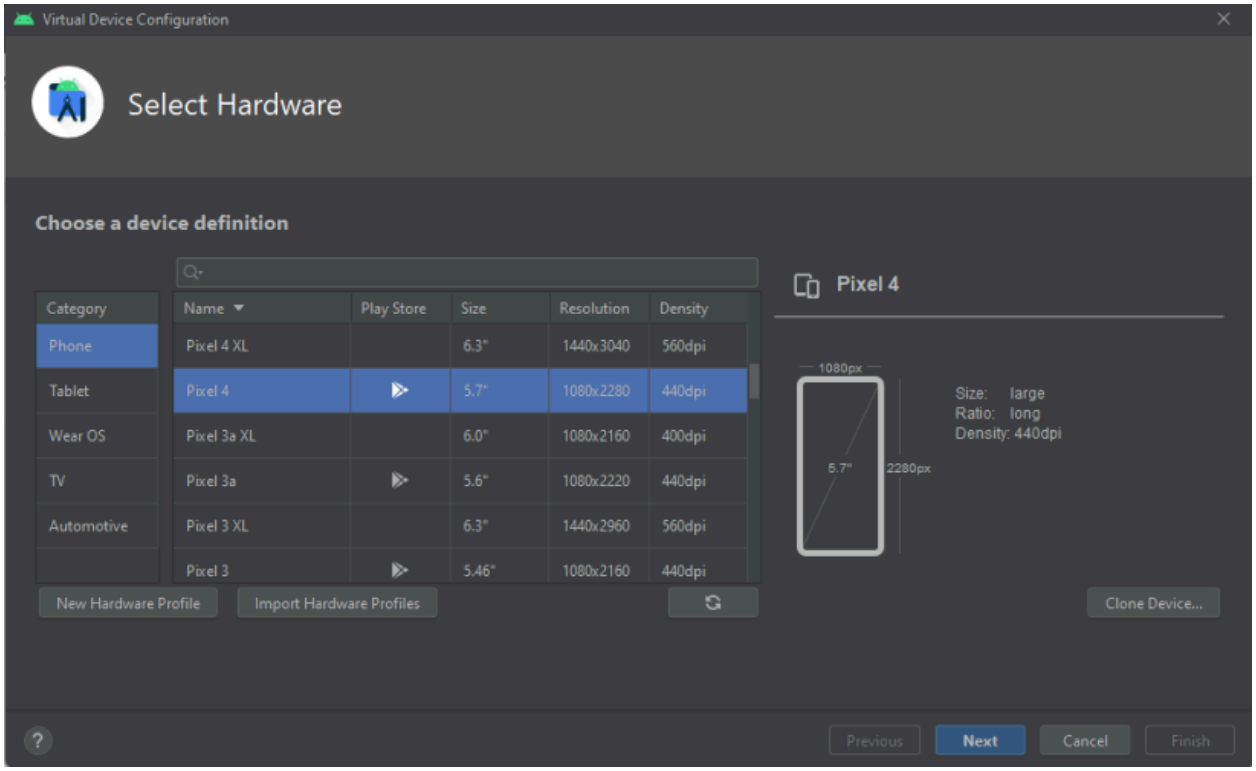

Android Studio Setup Wizard
Welcome
Android Studio
Welcome! This wizard will set up your development environment for Android Studio.
Additionally, the wizard will help port existing Android apps into Android Studio
or create a new Android application project.
Previ...
Next
Cancel
Finish


Android Studio Setup Wizard
Downloading Components
Preparing "Install SDK Patch Applier v4 (revision: 1)".
Downloading https://dl.google.com/android/repository/3534162-studio.sdk-patcher.zip
"Install SDK Patch Applier v4 (revision: 1)" ready.
Installing SDK Patch Applier v4 in
C:\Users\INFT505-19\AppData\Local\Android\Sdk\patcher\v4
"Install SDK Patch Applier v4 (revision: 1)" complete.
"Install SDK Patch Applier v4 (revision: 1)" finished.
Preparing "Install Intel x86 Emulator Accelerator (HAXM installer) (revision:
7.6.5)".
Downloading https://dl.google
.com/android/repository/extras/intel/haxm-windows_v7_6_5.zip
"Install Intel x86 Emulator Accelerator (HAXM installer) (revision: 7.6.5)" ready.
Installing Intel x86 Emulator Accelerator (HAXM installer) in
C:\Users\INFT505-19\AppData\Local\Android\Sdk\extras\intel
Previ...
Next
Cancel
Finish

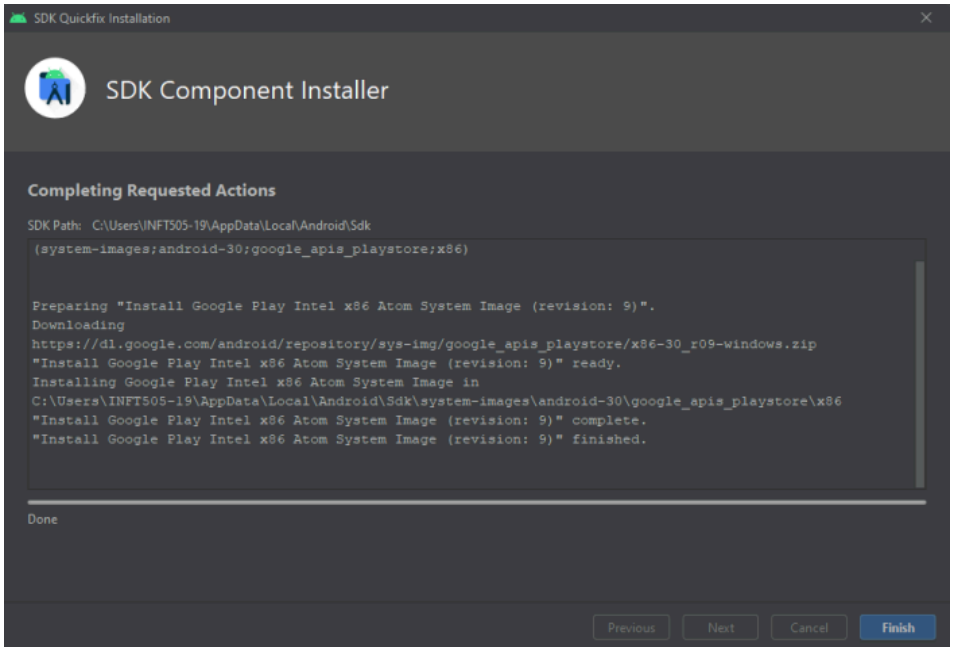
Step 7.1: To set an Android emulator, go to Android Studio > Tools > SDK Manager and select Create Virtual Device. You will get the following screen:



Step 7.2: Choose your device definition and click on Next.



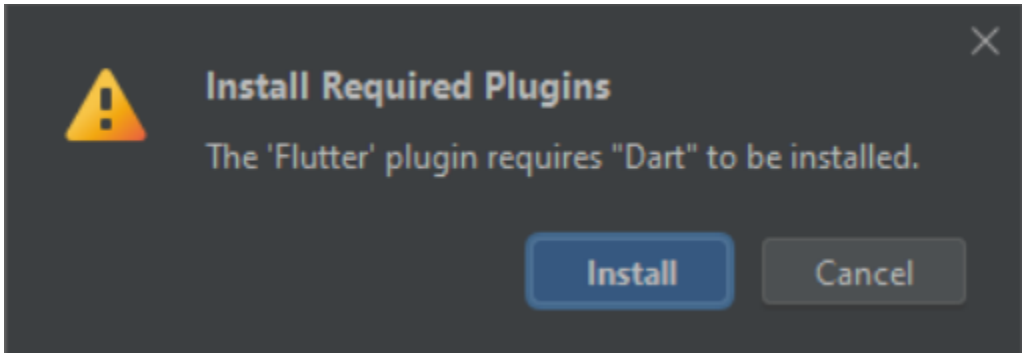
Step 7.3: Select and download the latest operating system for our Emulator and click on Finish.



Step 7.4: Click on the Run button and the following screen will be displayed.

Step 8: Now, install Flutter and Dart plugins for building the Flutter application in Android Studio. These plugins provide a template to create a Flutter application and give the option to run and debug the Flutter application in the Android Studio itself.

Open the Android Studio and then go to File->Settings->Plugins. Now, search the Flutter plugin. If found, select the Flutter plugin and click install. When you click on install, it will ask you to install the Dart plugin as shown below screen. Click Install to proceed.



Finally when all these Steps are followed restart the Android Studio once and then your Flutter environment is successfully configured.

Conclusion: In conclusion, the installation and configuration of the Flutter environment are crucial steps that set the foundation for efficient and seamless app development. By following the platform-specific installation instructions provided by Flutter, developers can ensure they have the necessary tools and dependencies in place.

EXPERIMENT NO.4

Experiment No 4 **To create an interactive Form using Form widget**	
ROLL NO	
NAME	
CLASS	**D15-B**
SUBJECT	**MAD & PWA Lab**
LO-MAPPE D	

Aim: To create an interactive Form using Form widget

Theory:
Flutter Form is a mechanism for capturing and validating user input within a set of input fields. It's constructed using the Form widget, which acts as a container for multiple FormField widgets. FormField widgets represent individual input fields such as TextFormField or DropdownButtonFormField.

Username Validator:
The username validator ensures that the username field is not left empty. This is crucial for creating a unique identifier for each user. Users are prompted to enter a username, and if the field is left blank, a validation error is displayed, prompting the user to fill in the required information.

Email Validator:
The email validator ensures that the email entered by the user follows a valid email format. It checks if the email field is empty and then uses a regular expression pattern to validate the email format. This validation ensures that users provide a properly formatted email address, helping maintain communication integrity within the application.

Password Validator:
The password validator ensures that the password provided by the user meets certain criteria, such as a minimum length requirement. In this case, the validator checks if the password field is empty and whether the length of the password is at least 8 characters long. This validation helps enhance security by ensuring that users create strong passwords to protect their accounts.

Flutter Validation:
Validation in TextFormField is performed by providing a validator function to the validator property. This function takes the current value of the input field and returns a String error message if the input is invalid, or null if the input is valid. When the form is submitted, each TextFormField's validator function is invoked to check the validity of the input.

Submit Button:
The Submit button in a Flutter form is typically implemented using a button widget, such as ElevatedButton or TextButton. When pressed, the Submit button triggers the form submission process. Before submitting the form, the form's state is checked using the GlobalKey associated with the Form. If the form's state is valid, the form data is processed or submitted to a backend server. If the form's state is invalid, the user is notified of any validation errors, and the submission is prevented.

Code:

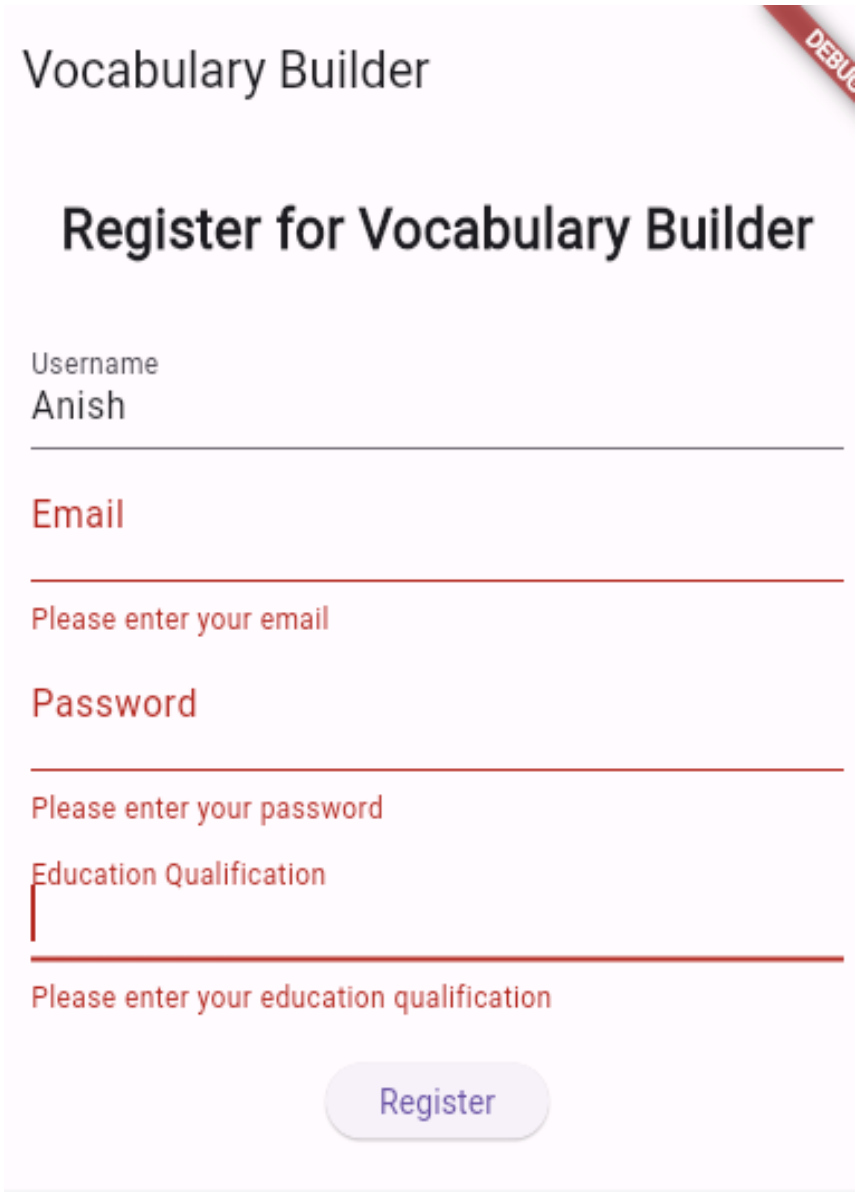
```
import 'package:flutter/material.dart';

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

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

class RegisterPage extends StatefulWidget {
  @override
  _RegisterPageState createState() => _RegisterPageState();
}

class _RegisterPageState extends State<RegisterPage> {
  final _formKey = GlobalKey<FormState>();

  String _username = "";
  String _email = "";
  String _password = "";
  String _educationQualification = "";

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: Text('Vocabulary Builder'),
      ),
      body: Form(
        key: _formKey,
        autovalidateMode: AutovalidateMode.onUserInteraction,
        child: Padding(
          padding: const EdgeInsets.all(20.0),
```

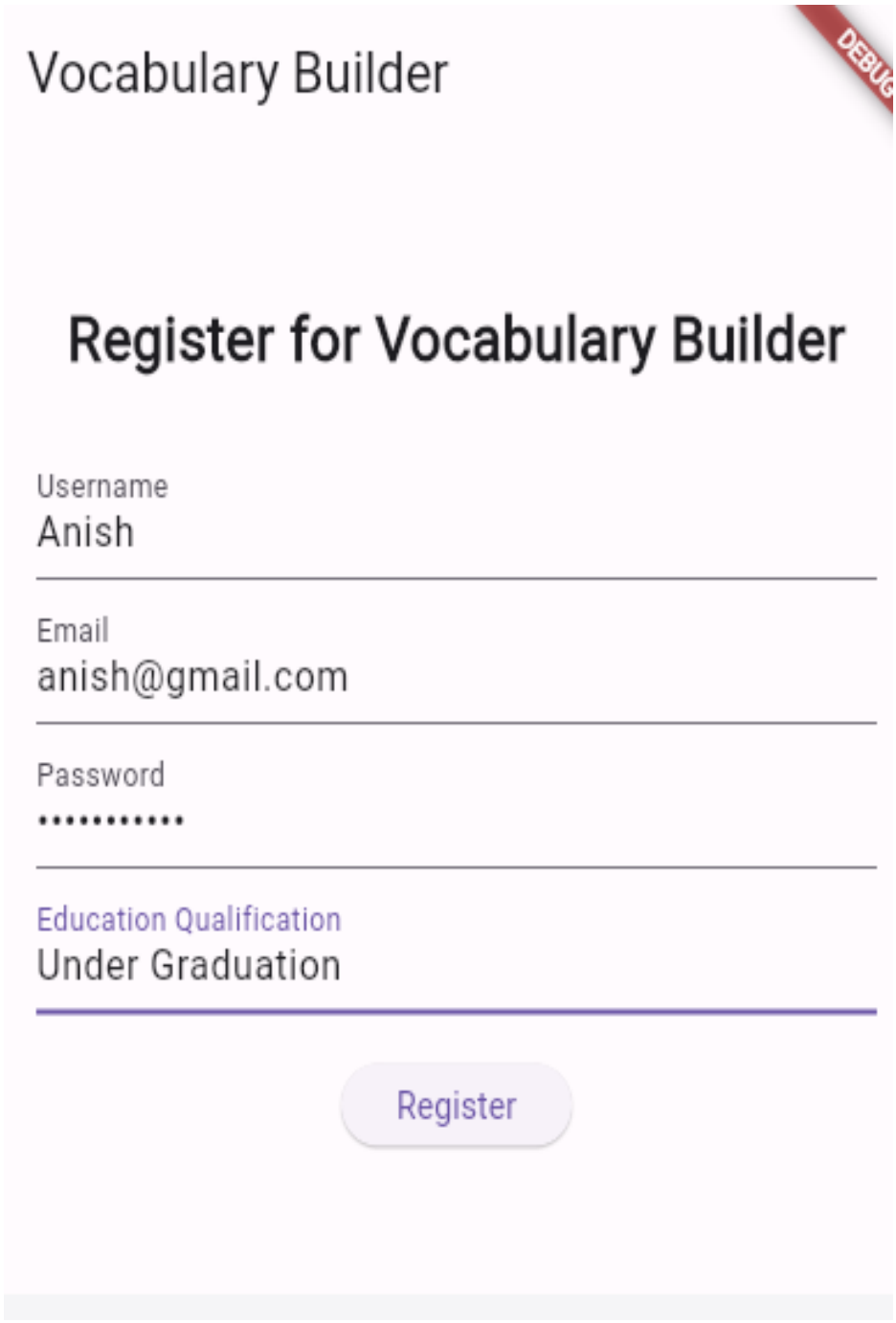
```
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Center(
              child: Text(
                'Register for Vocabulary Builder',
                style: TextStyle(fontSize: 24, fontWeight: FontWeight.bold),
              ),
            ),
            SizedBox(height: 20.0),
            TextFormField(
              decoration: InputDecoration(labelText: 'Username'),
              validator: (value) {
                if (value!.isEmpty) {
                  return 'Please enter your username';
                }
                return null;
              },
              onSaved: (value) => _username = value!,
            ),
            TextFormField(
              keyboardType: TextInputType.emailAddress,
              decoration: InputDecoration(labelText: 'Email'),
              validator: (value) {
                if (value!.isEmpty) {
                  return 'Please enter your email';
                } else if
(!RegExp(r"^[a-zA-Z0-9.a-zA-Z0-9.!#$%&'*+/=?^_`{|}~-]+@[a-zA-Z0-9-]+\.[a-zA-Z]+").hasMatch(val
ue!)) {
                  return 'Please enter a valid email address';
                }
                return null;
              },
              onSaved: (value) => _email = value!,
            ),
            TextFormField(
              obscureText: true,
              decoration: InputDecoration(labelText: 'Password'),
              validator: (value) {
                if (value!.isEmpty) {
                  return 'Please enter your password';
                } else if (value!.length < 8) {
                  return 'Password must be at least 8 characters';
                }
```

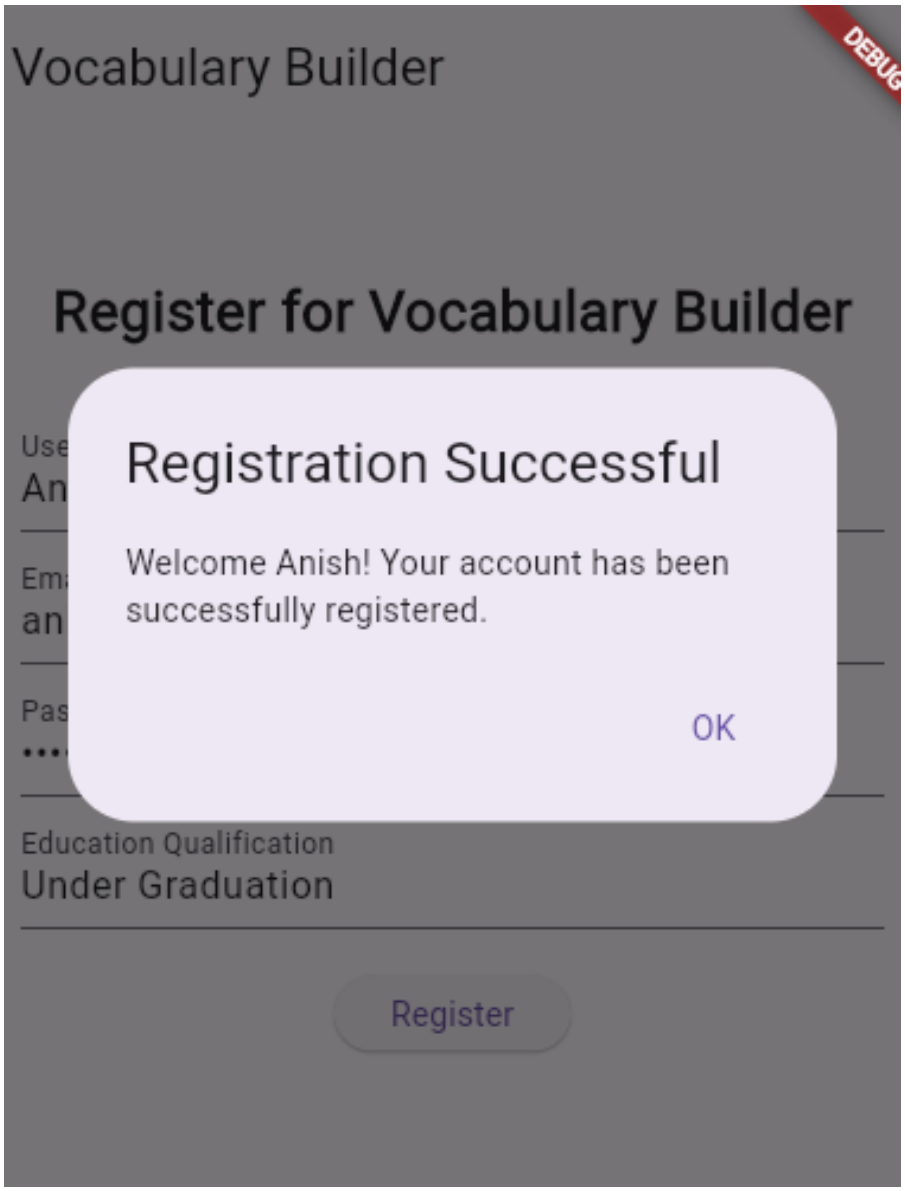
```
    return null;
  },
  onSaved: (value) => _password = value!,
),
TextFormField(
  decoration: InputDecoration(labelText: 'Education Qualification'),
  validator: (value) {
    if (value!.isEmpty) {
      return 'Please enter your education qualification';
    }
    return null;
  },
  onSaved: (value) => _educationQualification = value!,
),
SizedBox(height: 20.0),
ElevatedButton(
  onPressed: () {
    if (_formKey.currentState!.validate()) {
      _formKey.currentState!.save();
      // Handle registration logic here (e.g., send data to server)
      showDialog(
        context: context,
        builder: (BuildContext context) {
          return AlertDialog(
            title: Text('Registration Successful'),
            content: Text('Welcome $_username! Your account has been successfully registered.'),
            actions: [
              TextButton(
                onPressed: () {
                  Navigator.of(context).pop();
                  Navigator.push(
                    context,
                    MaterialPageRoute(builder: (context) => WelcomePage()),
                  );
                },
                child: Text('OK'),
              ),
            ],
          );
        },
      );
    }
  },
  child: Text('Register'),
```

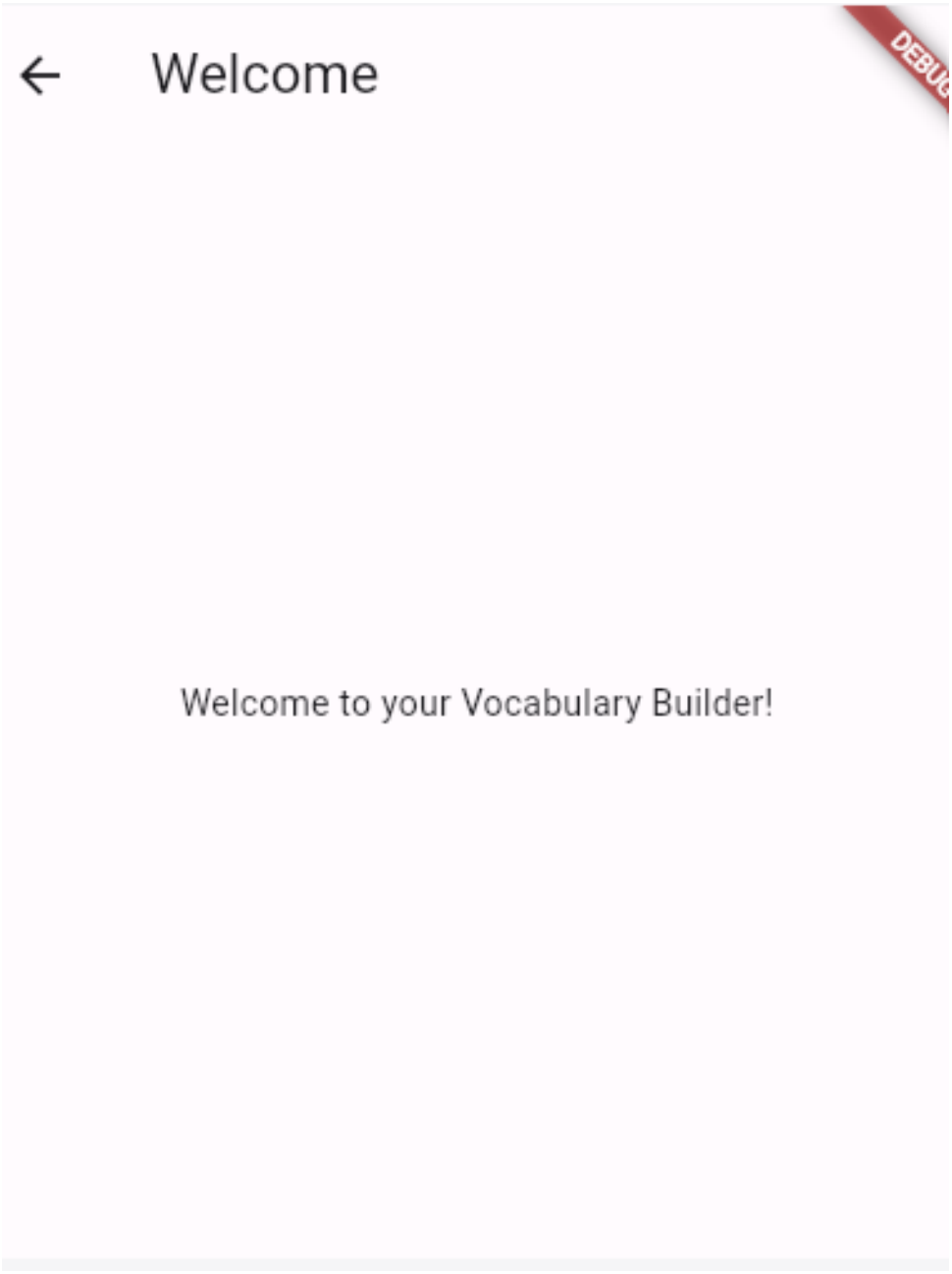
```
            ),
          ],
        ),
      ),
    ),
  );
 }
}

class WelcomePage extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(
    title: Text('Welcome'),
   ),
   body: Center(
    child: Text('Welcome to your Vocabulary Builder!'),
   ),
  );
 }
}
```

Output:


Vocabulary Builder
Register for Vocabulary Builder
Username
Anish
Email
Please enter your email
Password
Please enter your password
Education Qualification
Please enter your education qualification
Register


Vocabulary Builder
DEBUG
Register for Vocabulary Builder
Username
Anish
Email
anish@gmail.com
Password
Education Qualification
Under Graduation
Register


Vocabulary Builder
DEBUG
Register for Vocabulary Builder
Registration Successful
Welcome Anish! Your account has been successfully registered.
OK
Education Qualification
Under Graduation
Register



Conclusion:

Flutter Form is a powerful mechanism for capturing and validating user input using widgets such as TextFormField, GlobalKey, and button widgets. It allows developers to create interactive forms with custom validation logic and submit user data efficiently.

Name: Mohit Ailani **Roll No:2** **Div: D15B**

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:

Progressive Web Apps (PWAs) are a type of web application that combines the best features of web and mobile apps to offer users a seamless and engaging experience. Progressive Web Apps (PWAs) represent a new paradigm in web development, seamlessly blending the strengths of web technologies with the immersive user experience of native mobile apps. By leveraging responsive design principles, service workers for offline functionality, and the Web App Manifest for installation and customization, PWAs deliver fast, reliable, and engaging experiences across devices and network conditions. With features like push notifications, homescreen installation, and seamless navigation, PWAs bridge the gap between web and native apps, offering users a frictionless journey and empowering developers to build versatile, accessible, and modern web applications.

Overview of PWAs:

1. Responsive Design: PWAs are designed to be responsive, meaning they can adapt andprovide a great user experience across various devices, including desktops, tablets, and smartphones.

2. Progressive Enhancement: PWAs are built using progressive enhancement principles. This means they should work for all users, regardless of the browser or device they use. Advanced features are progressively enhanced for users with modern browsers and devices.

3. App-Like Experience: PWAs aim to provide an app-like experience, including smooth navigation, offline functionality, push notifications, and the ability to be added to the home screen.

4. Service Workers: One of the key technologies behind PWAs is service workers. These are scripts that run in the background and enable features like offline caching, background synchronization, and push notifications.

5. Web App Manifest: The Web App Manifest is a JSON file that contains metadata about the PWA, such as its name, icons, start URL, display mode, theme colors, and more. This manifest file is used by browsers to install the PWA and customize its appearance on the user's device.

6. Offline Functionality: PWAs can work offline or in low-connectivity environments by caching assets and data using service workers. This ensures that users can still access content and perform actions even when they are not connected to the

7. Fast and Reliable: PWAs are designed to be fast and reliable, providing quick load times and smooth interactions to enhance the user experience.

8. Secure: PWAs are served over HTTPS to ensure data security and protect user privacy,especially when dealing with sensitive information or transactions.

9. Cross-Platform Compatibility: PWAs are platform-agnostic and can run on various operating systems and browsers, reducing development effort and reaching a broader audience.

Overall, PWAs combine the reach and accessibility of the web with the capabilities and engagement of native apps, making them a compelling choice for modern web development.

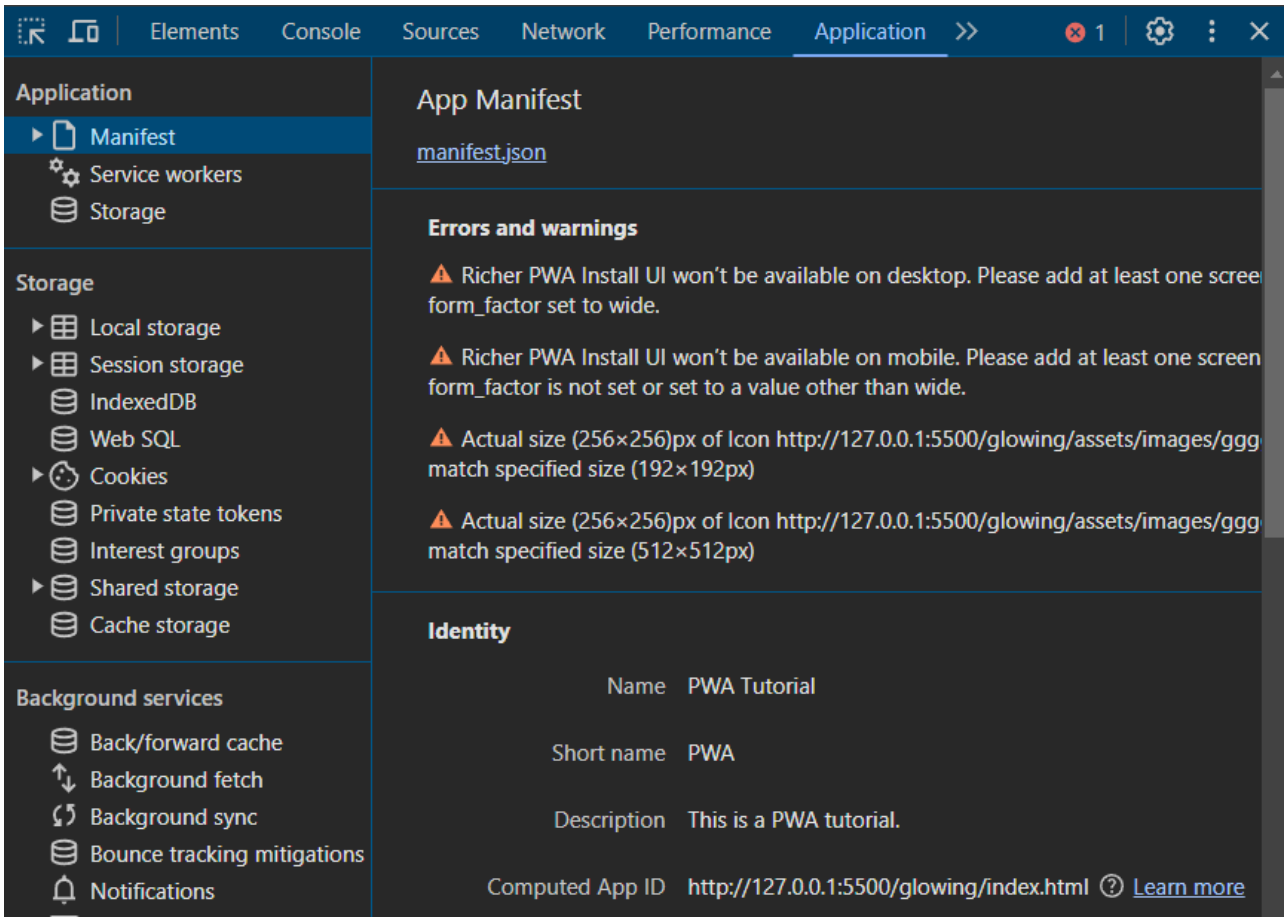
Implementation:

1. Create an HTML file for your ecommerce website that will contain a link to the manifest.json file.

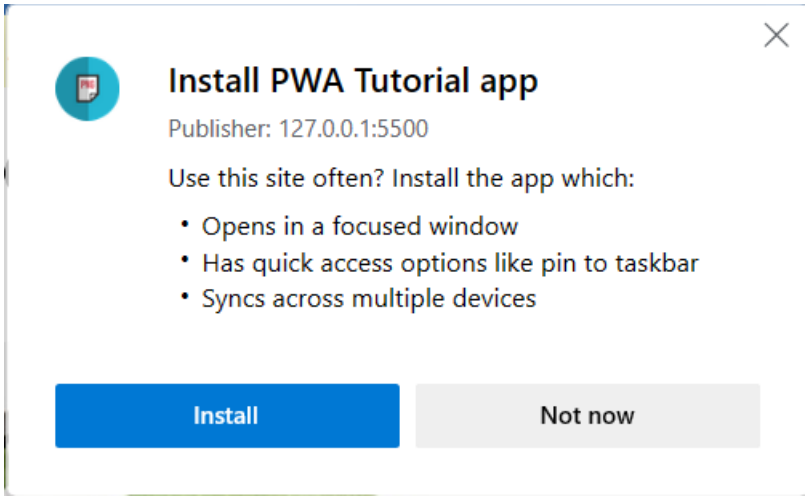
```
<head>
  <meta charset="UTF-8">
  <meta http-equiv="X-UA-Compatible" content="IE=edge">
  <meta name="viewport" content="width=device-width, initial-scale=1.0">
  <link rel="manifest" href="manifest.json">
  <title>Glowing - Reveal The Beauty of Skin</title>
```

2. Create a manifest.json() file in the same directory. This file basically contains information aboutthe web application. Some basic information includes the application name, starting URL, theme color, and icons. All the information required is specified in the JSON format. The source and size of the icons are also defined in this file.

3. Open the index.html file in Chrome navigate to the Application Section in the Chrome Developer Tools. Open the manifest column from the list.

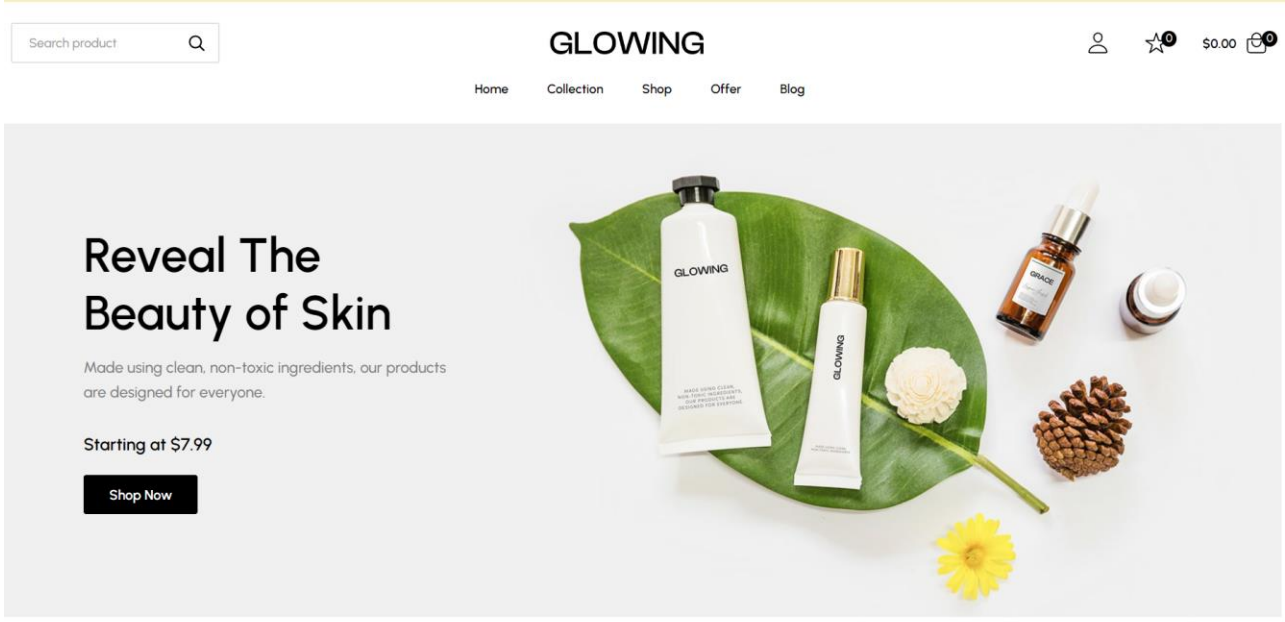


4. Installing the web application using the Install app button



5. Click on the install button to install the application. The application would then be installed, and it would be visible on the desktop.

Final Application:



Conclusion:

Hence we have understood and studied the working of Progressive web applications and the features of Progressive web applications. Also, we have implemented the add to homescreen feature in Progressive web applications on oure-commerce website. By crafting a well-structured manifest.json() file with accurate metadata properties such as name, description, icons, and colors, developers can enhance the accessibility and user experience of their PWAs.

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.

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('/ser
  vice-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/servic
e-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:

//Js code in index.html file

```
<script>
  window.addEventListener('load', () => {
  registerSW();
  });
```

```
    // Register the Service Worker
    async function registerSW() {
      if ('serviceWorker' in navigator) {
        try {
          await navigator.serviceWorker.register('/sw.js', { scope: '/index.html' });
          console.log('SW registration successful');
        } catch (error) {
          console.log('SW registration failed:', error);
        }
      } else {
        console.log('Service Worker is not supported in this browser.');
      }
    }
  </script>
```

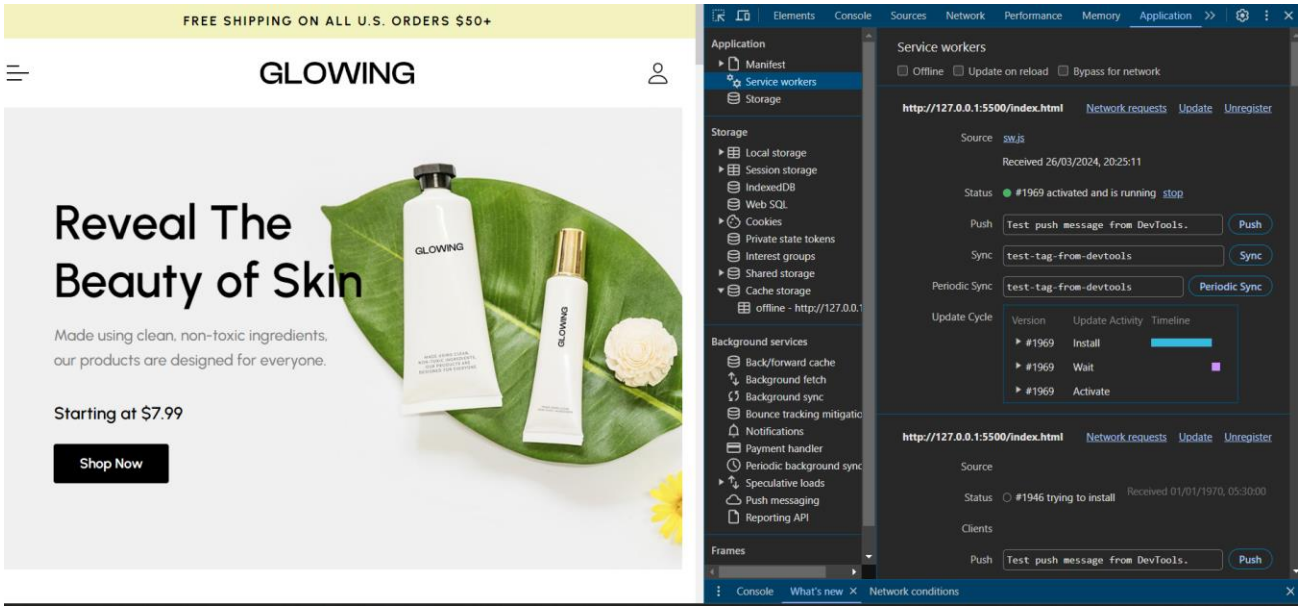
//serviceworker.js file

```
var staticCacheName = "pwa-v1"; // Update cache name with versioning
self.addEventListener("install", function (event) {
  event.waitUntil(preLoad());
});

var filesToCache = ["/index.html"];

var preLoad = function () {
  return caches.open("offline").then(function (cache) {
    // caching index and important routes
    return cache.addAll(filesToCache);
  });
};

self.addEventListener("fetch", function (event) {
  event.respondWith(
    checkResponse(event.request).catch(function () {
      return returnFromCache(event.request);
    })
  );
  event.waitUntil(addToCache(event.request));
});

var checkResponse = function (request) {
  return new Promise(function (fulfill, reject) {
    fetch(request).then(function (response) {
      if (response.status !== 404) {
        fulfill(response);
      } else {
        reject();
      }
    }, reject);
  });
};
```

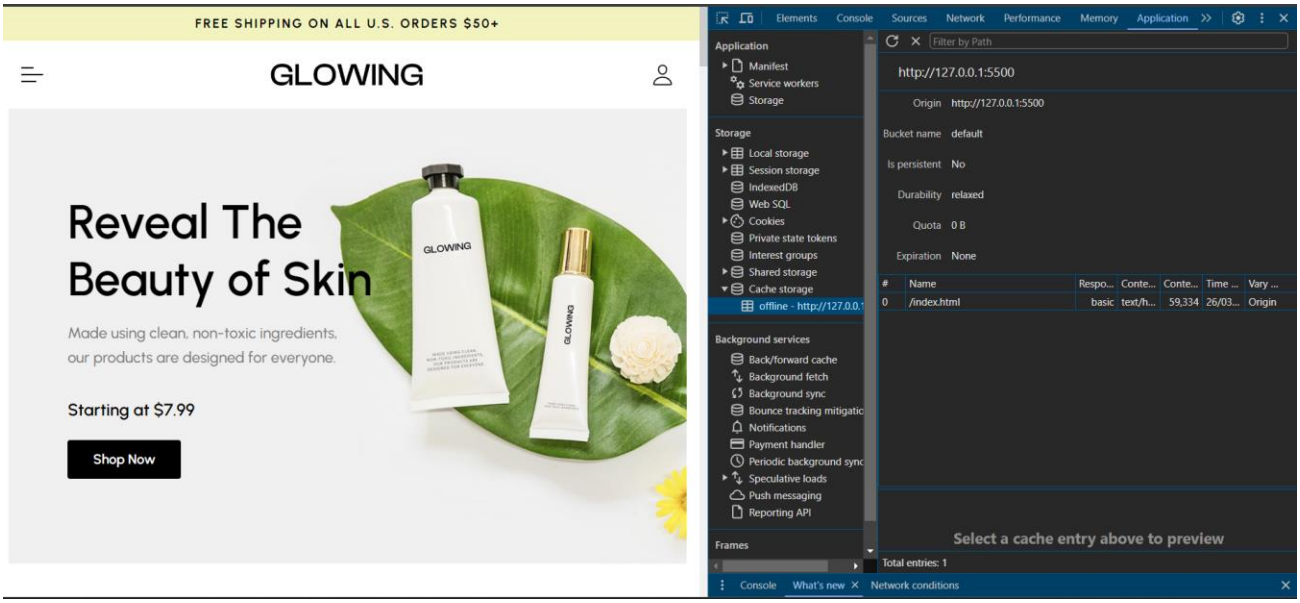
```
var addToCache = function (request) {
  return caches.open("offline").then(function (cache) {
    return fetch(request).then(function (response) {
      return cache.put(request, response);
    });
  });
};

var returnFromCache = function (request) {
  return caches.open("offline").then(function (cache) {
    return cache.match(request).then(function (matching) {
      if (!matching || matching.status == 404) {
        return cache.match("offline.html");
      } else {
        return matching;
      }
    });
  });
};
```

Output:

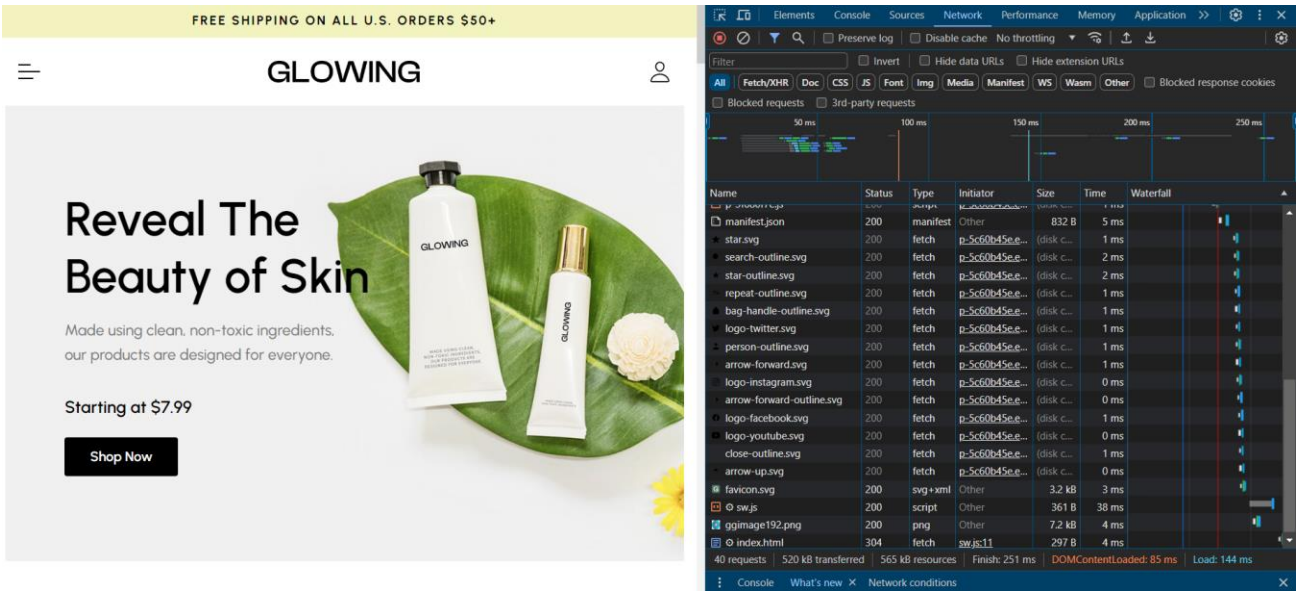
1.Cache



2.Serviceworker



3.Offline



Conclusion: In conclusion, testing the offline functionality of a Progressive Web App (PWA) is crucial to ensure a reliable user experience, especially in scenarios where internet connectivity may be unreliable or unavailable. By simulating an offline environment and thoroughly testing the behavior of the app, including service worker registration, caching of resources, and progressive enhancement, developers can ensure that the app continues to function seamlessly even without an internet connection.

PWA EXPERIMENT NO.10

AIM: To study and implement deployment of Ecommerce PWA to GitHub Pages.

THEORY:

GitHub Pages is a web hosting service provided by GitHub, allowing users to host static websites directly from their GitHub repositories. Here are some key theoretical aspects and concepts related to GitHub Pages:

1. **Static Websites:** GitHub Pages is designed for hosting static websites, which means that the content of the website is fixed and doesn't change dynamically based on user interactions or server-side processing. This makes GitHub Pages ideal for blogs, portfolios, documentation sites, and simple informational websites.

2. **Version Control:** GitHub Pages integrates seamlessly with Git, the version control system used by GitHub. This allows developers to track changes to their website code, collaborate with others using branches and pull requests, and revert to previous versions if needed.

3. **Branches:** GitHub Pages can be configured to publish different branches of a repository. For example, the `master` branch is commonly used for the production version of the website, while other branches can be used for testing, development, or feature experimentation without affecting the live site.

4. **Custom Domains:** Users can map a custom domain (e.g., `www.example.com`) to their GitHub Pages site, giving it a more professional and branded appearance. This involves configuring DNS settings and updating the GitHub Pages settings accordingly.

5. **Supported Technologies:** GitHub Pages supports various technologies for building static websites, including HTML, CSS, JavaScript, and static site generators like Jekyll. This flexibility allows developers to choose the tools and frameworks that best suit their needs.

6. **Continuous Integration:** GitHub Pages can be integrated with continuous integration (CI) tools like GitHub Actions, allowing for automated testing, building, and deployment of the website whenever changes are pushed to the repository. This helps maintain a smooth development workflow and ensures that the live site is always up-to-date.

7. **Free Hosting:** One of the major advantages of GitHub Pages is that it provides free hosting for static websites, making it accessible to individual developers, small projects, open-source initiatives, and educational purposes without incurring hosting costs.

8. **Community and Support:** GitHub Pages is supported by a large community of

developers and has extensive documentation and resources available. This makes it easy for users to find help, troubleshoot issues, and learn best practices for optimizing their GitHub Pages sites.

By understanding these theoretical aspects, developers can effectively utilize GitHub Pages to host and manage their static websites, leveraging its features for version control, collaboration, automation, and cost-effective hosting.

Pros of using GitHub Pages :

1. Free hosting for static websites.

2. Seamless integration with Git for version control.

3. Custom domain mapping and community support.

Cons of using GitHub Pages:

1. Limited functionality for dynamic content.

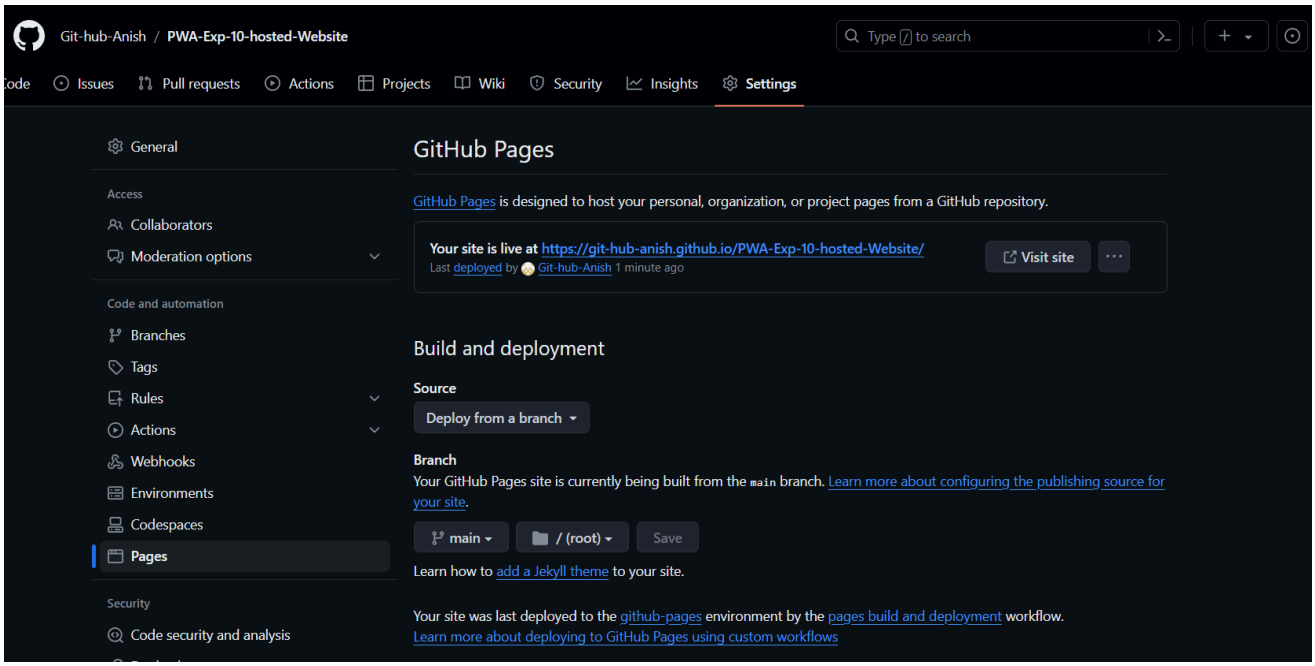
2. Storage limits and dependency on GitHub's infrastructure.

3. Not suitable for large-scale applications with complex features.

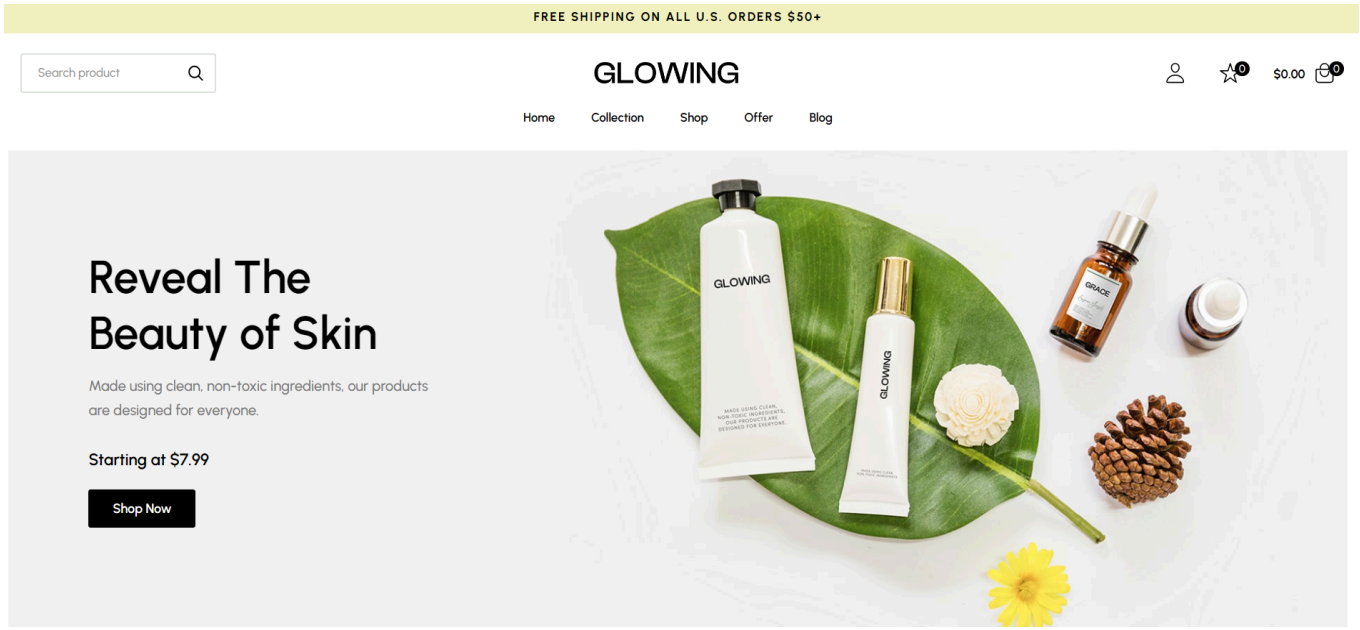
Link to the GitHub repository:
https://github.com/Anish-Mayekar/MLP-MAD-Lab/tree/main/PWA-Experiment-No-10/glowing

Hosted website link: **https://git-hub-anish.github.io/PWA-Exp-10-hosted-Website/**

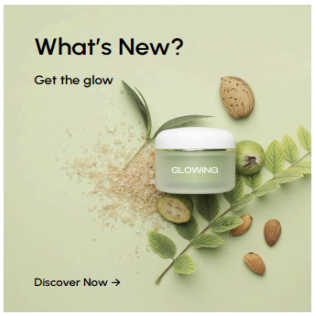
GitHub pages Screenshot:



Hosted website Screenshots:



Home Collection Shop Offer Blog



Our Bestsellers

Shop All Products →

Home Collection Shop Offer Blog

SPECIAL OFFER -20%

Mountain Pine Bath Oil

Made using clean, non-toxic ingredients, our products are designed for everyone.

15 : 21 : 46 : 08

Get Only $39.00

CONCLUSION : Hence we have deployed our E-commerce Progressive web application via Github pages and understood the working of GitHub pages.

Experiment 11

Aim: To use google Lighthouse PWA Analysis Tool to test the PWA functioning.

Theory:

Google Lighthouse is an open-source, automated tool for improving the quality of webpages. It is used to audit web pages, measure their performance, accessibility, progressive web app (PWA) capabilities, SEO (Search Engine Optimization), and more. Lighthouse is commonly used by developers, webmasters, and SEO professionals to identify areas where a website can be optimized for better user experience, performance, and search engine rankings.

Features of Google Lighthouse:

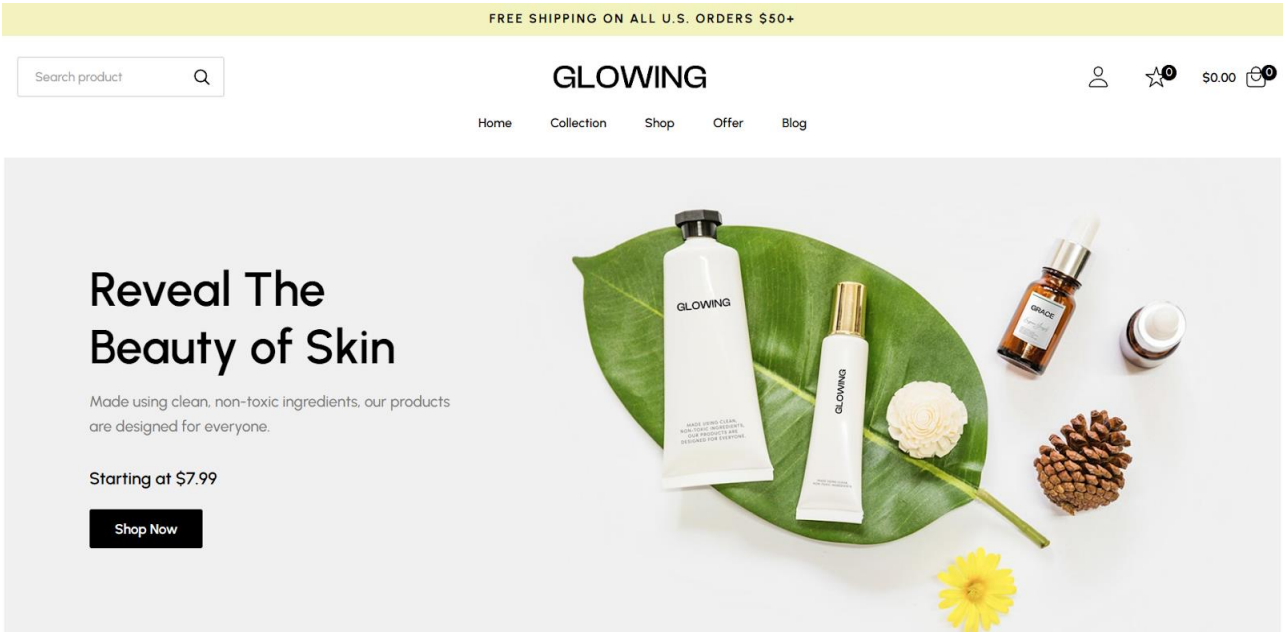
1. **Performance:**
    a. Lighthouse assesses a webpage's loading speed and responsiveness.
    b. It provides metrics such as First Contentful Paint (FCP), Largest Contentful Paint (LCP), Time to Interactive (TTI), and more.
    c. Suggestions are given to improve these metrics, like optimizing images, deferring offscreen images, minifying CSS and JavaScript, etc.
2. **Accessibility:**
    a. Lighthouse evaluates a webpage's accessibility for users with disabilities.
    b. It checks for proper HTML markup, ARIA attributes, color contrast, andother factors that impact accessibility.
    c. Suggestions are provided to improve accessibility, such as adding alt textto images, ensuring keyboard navigation, and using proper heading structure.
3. **Best Practices:**
    a. Lighthouse checks a webpage against web development best practices.
    b. It looks for deprecated JavaScript features, security vulnerabilities, outdated libraries, and more.
    c. Recommendations are given to follow modern web development practicesand ensure a secure website.
4. **SEO (Search Engine Optimization):**
    a. Lighthouse provides insights into a webpage's SEO performance.
    b. It checks for meta tags, heading structure, sitemap usage, and otherSEO-related elements.

c. Recommendations are offered to improve the webpage's visibility in search engine results.

5. **Progressive Web App (PWA):**
   a. For websites aiming to be Progressive Web Apps, Lighthouse evaluatestheir PWA capabilities.
   b. It checks for features such as offline functionality, fast loading times onrepeat visits, and app-like behavior.
   c. Suggestions are given to enhance the website's PWA features, improvinguser engagement and experience.
6. **Metrics and Scoring:**
   a. Lighthouse provides a comprehensive report with scores for each category (Performance, Accessibility, Best Practices, SEO, PWA).
   b. These scores give a quick overview of how well a webpage is performingin each area.
   c. Developers can use these scores to prioritize improvements and track progress over time.
7. **Command Line Interface (CLI):**
   a. Lighthouse can be used both through its web interface and as a command-line tool.
   b. The CLI allows for integration into automated build processes and continuous integration (CI) workflows.
   c. This makes it easy to run audits regularly, ensuring ongoing improvementsto web performance and quality.
8. **Chrome DevTools Integration:**
   a. Lighthouse is integrated into Google Chrome's DevTools.
   b. Developers can run audits directly from the browser, making it convenientto analyze and improve web pages during development.
   c. The DevTools integration offers real-time feedback and suggestions as developers make changes to their code.

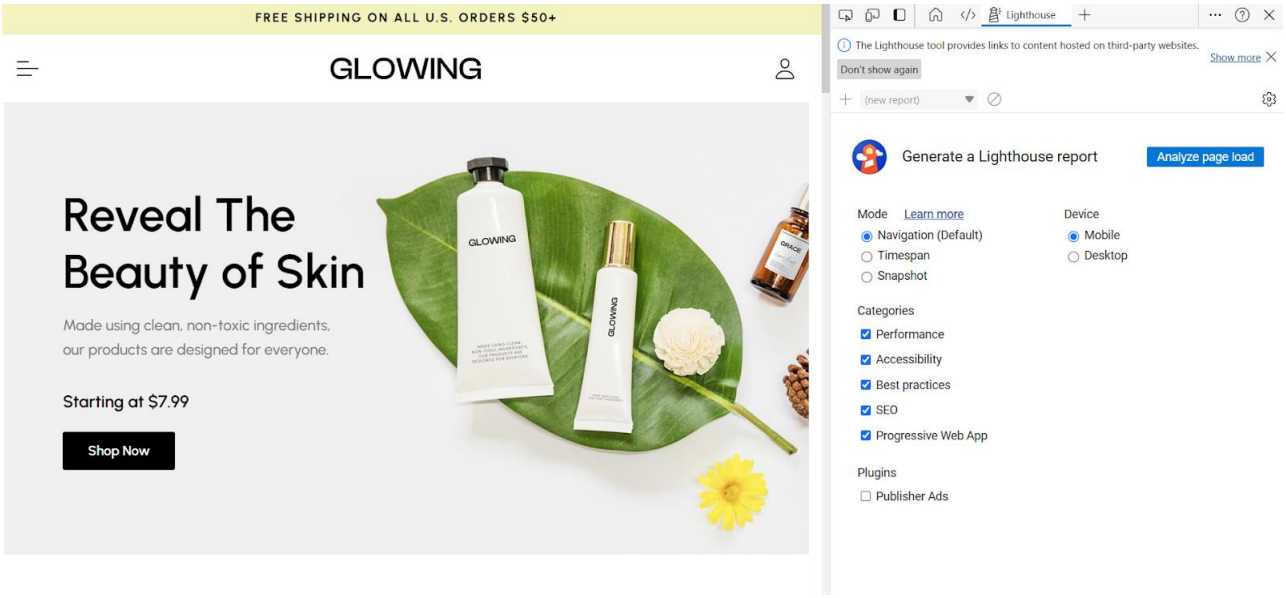
Steps: To use Google Lighthouse to test the Progressive Web App (PWA) functioning,follow these steps:

1. Open EDGE DevTools:
- Open edge browser.
- Go to the website you want to test as a PWA. - Right-click on the page and select "Inspect" or press `Ctrl+Shift+I` (Windows/Linux) or `Cmd+Option+I` (Mac) to openChrome DevTools.



2. Navigate to the Lighthouse Tab:

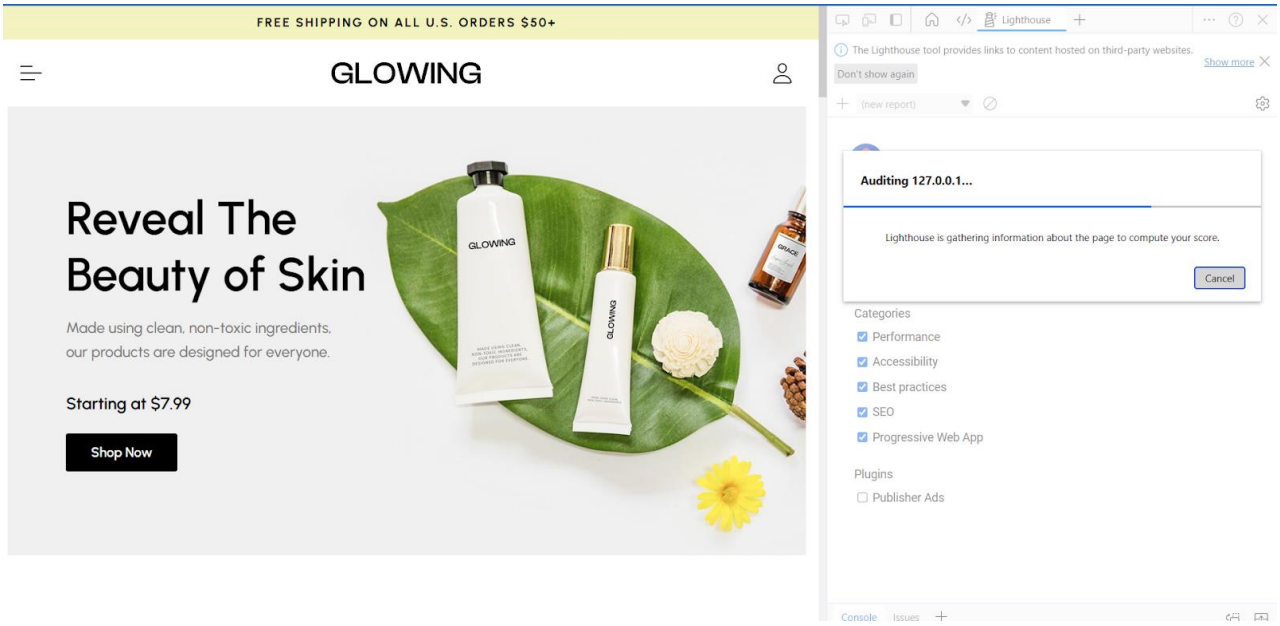
- In edge DevTools, click on the "Lighthouse" tab at the top of the DevTools panel.



3. Run the Lighthouse Audit:

- Click on the "Generate report" button to start the audit process.

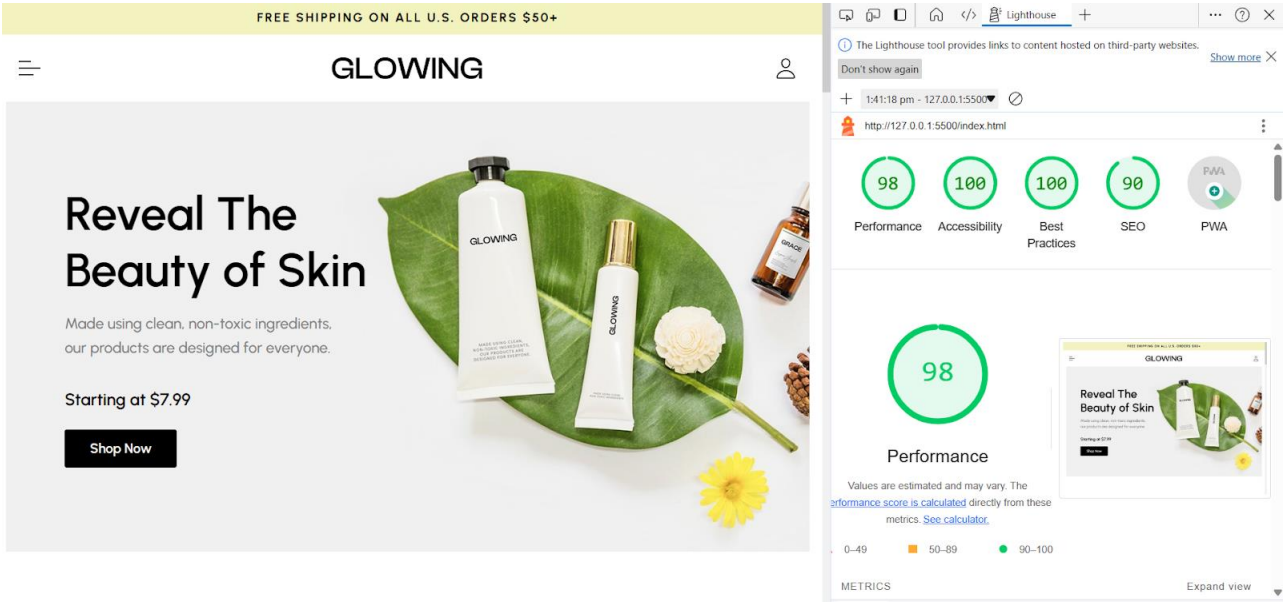
You can choose to audit for Performance, Accessibility, Best Practices, SEO, and Progressive Web App (PWA) functionality. Make sure to select the "Progressive WebApp" checkbox.



4. View the Audit Results:

-After the audit is complete, Lighthouse will display a report with scores and detailed information for each category.

-In the PWA section, you can check if your website meets the PWA criteria, such as having a service worker, being responsive on different devices, having a valid manifestfile, etc. - Lighthouse will provide suggestions for improvements and optimizations toenhance your PWA functionality.



CONCLUSION: Hence, we have understood the working of Google Lighthouse PWAanalysis tool and used Google Lighthouse tool to test and analyze the performancestatistics of our E-commerce Progressive web application.