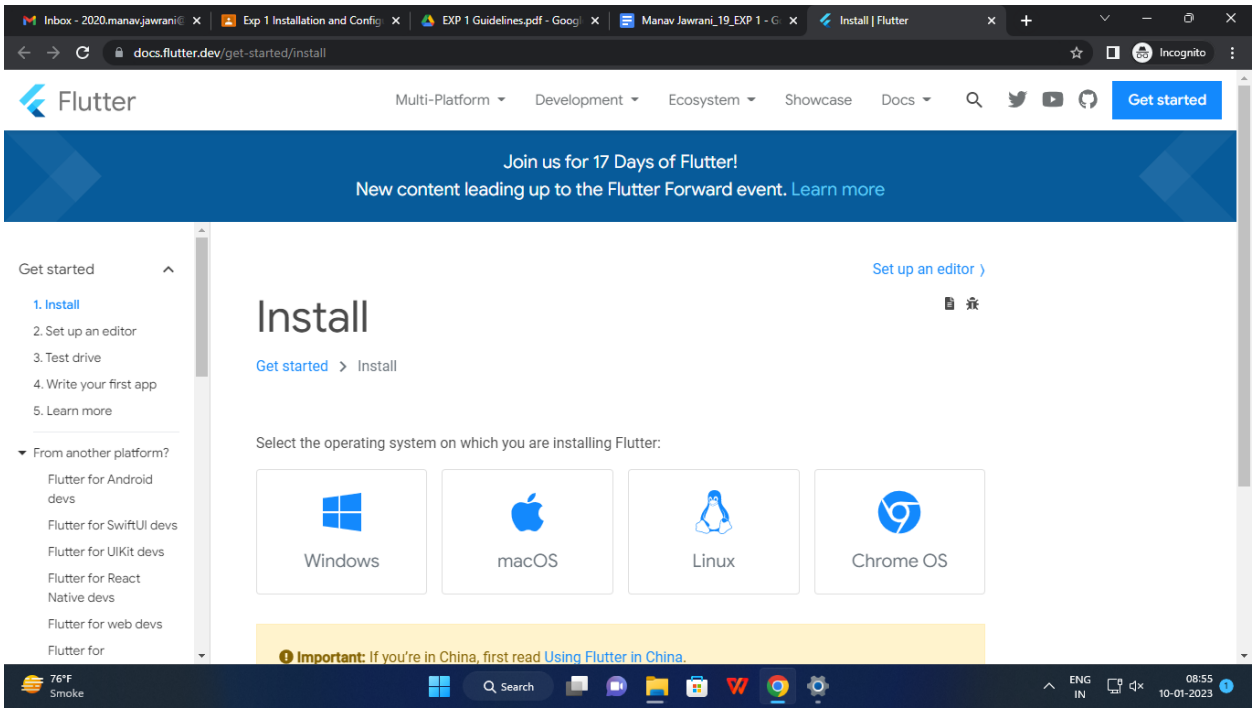
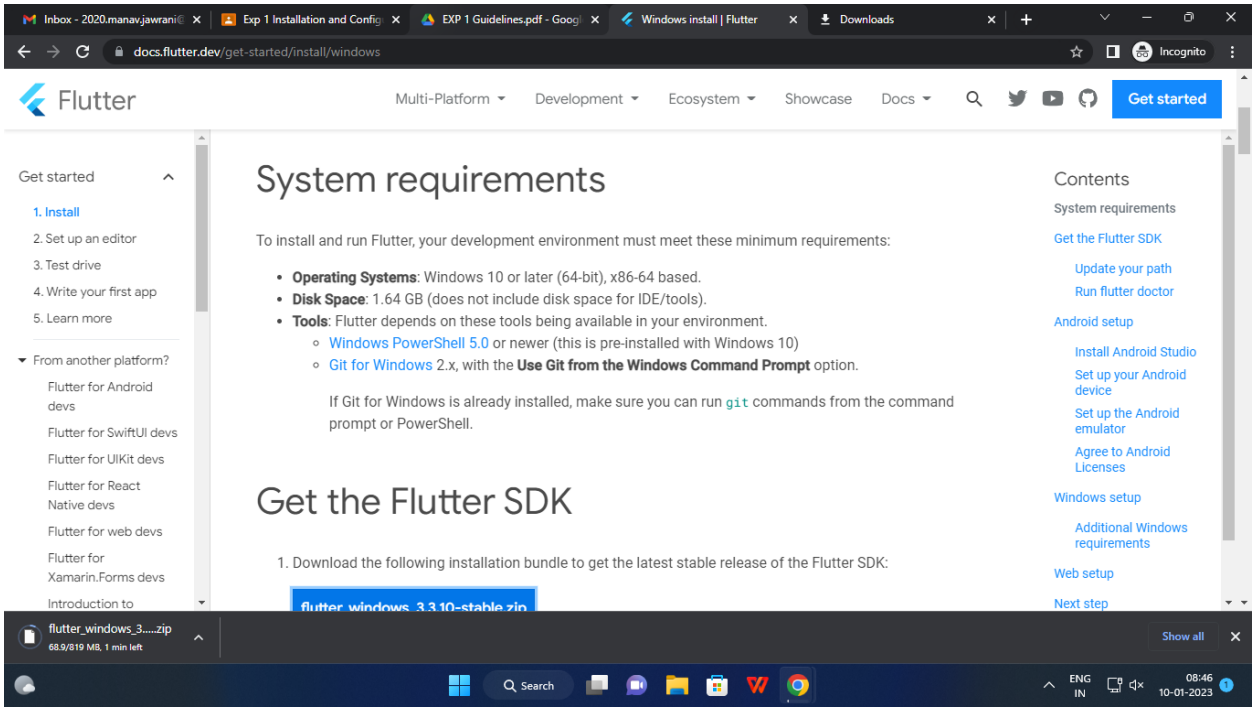
EXP 1: Installation and Configuration of Flutter Environment.

A. Install the Flutter SDK

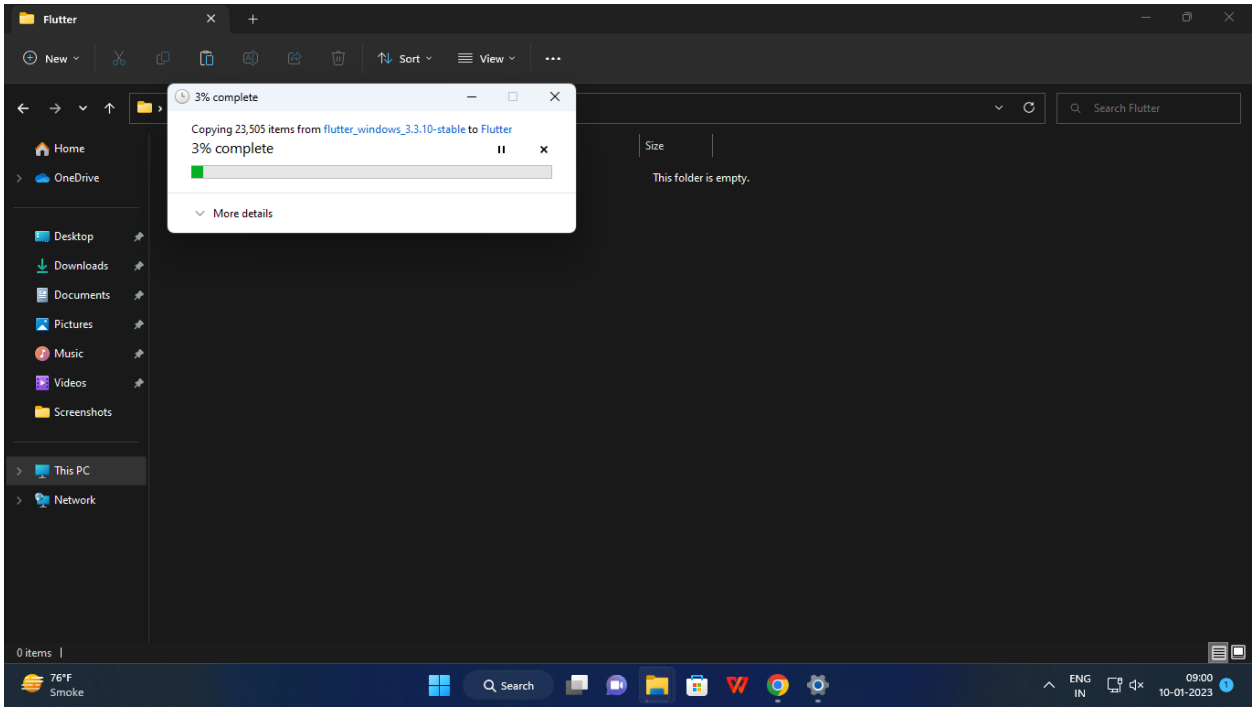
Step 1: Download the installation bundle of the Flutter Software Development Kit for windows. To download Flutter SDK, Go to its official website https://docs.flutter.dev/get-started/install , you will get the following screen.



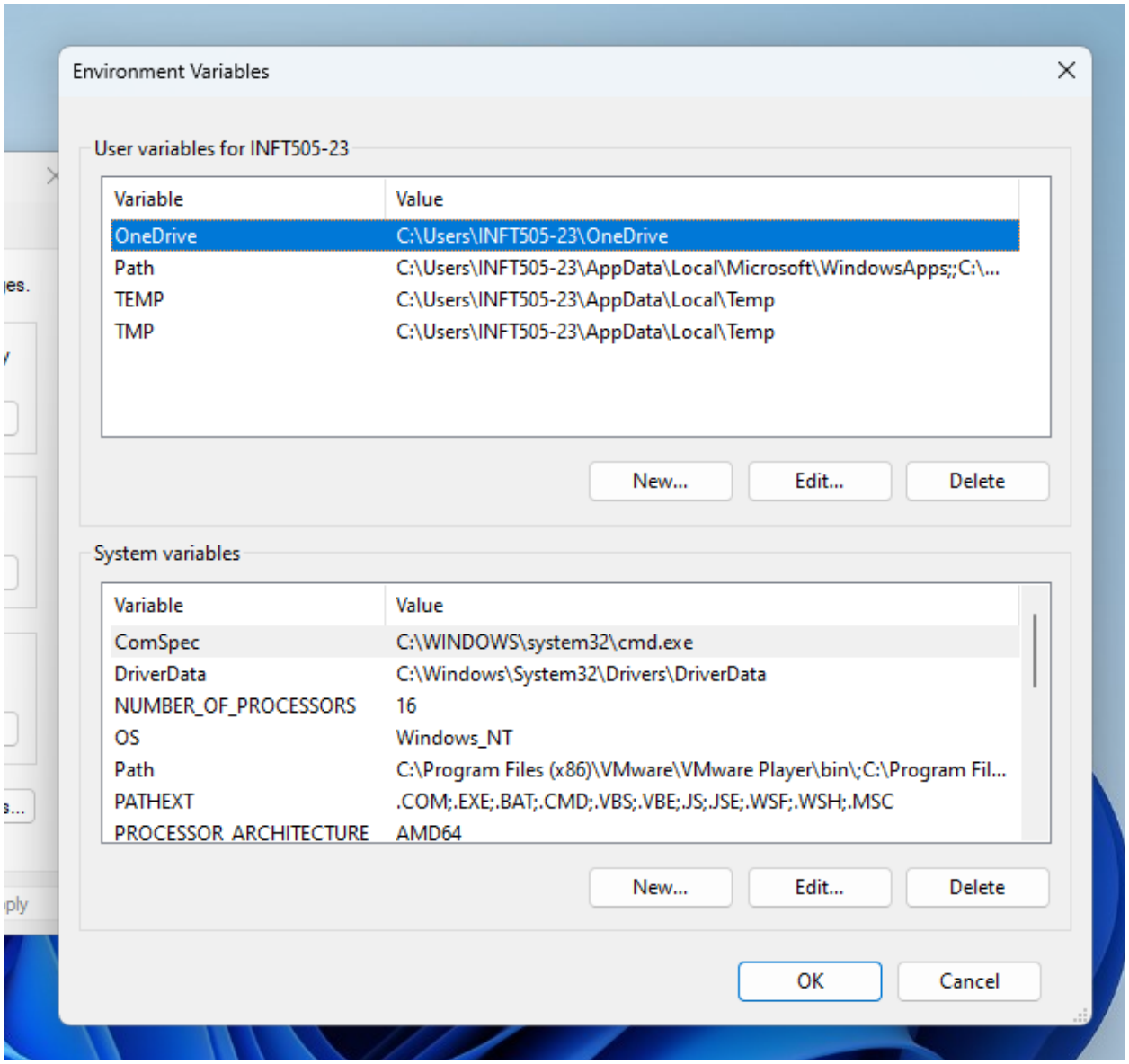
Step 2: Next, to download the latest Flutter SDK, click on the Windows icon. Here, you will find the download link for SDK.



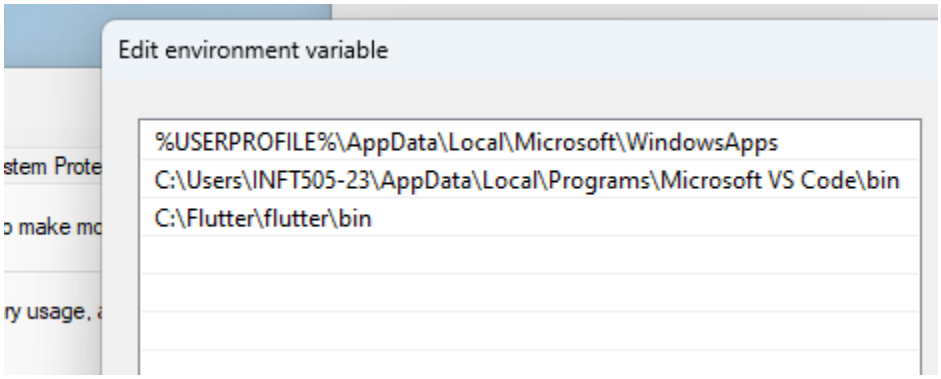
Step 3: When your download is complete, extract the zip file and place it in the desired installation folder or location, for example, C: /Flutter.



Step 4: To run the Flutter command in regular windows console, you need to update the system path to include the flutter bin directory. The following steps are required to do this:
Step 4.1: Go to MyComputer properties -> advanced tab -> environment variables. You will get the following screen.

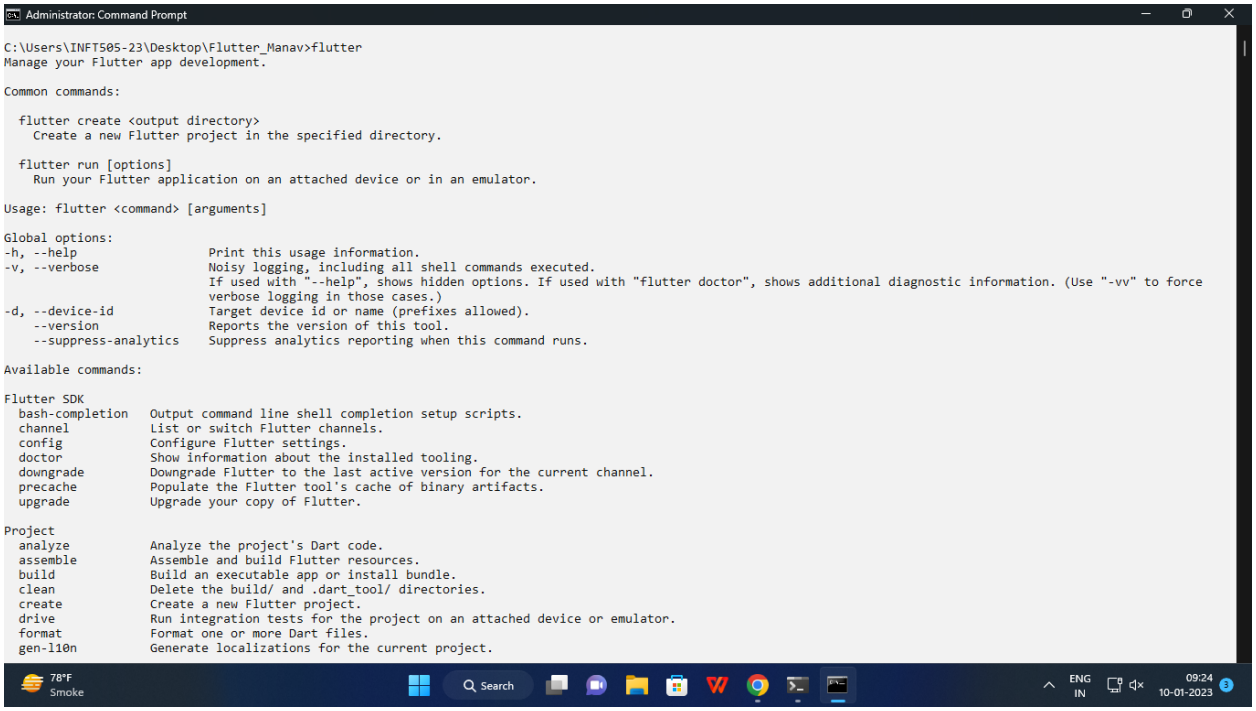


Step 4.2: Now, select path -> click on edit. The following screen appears

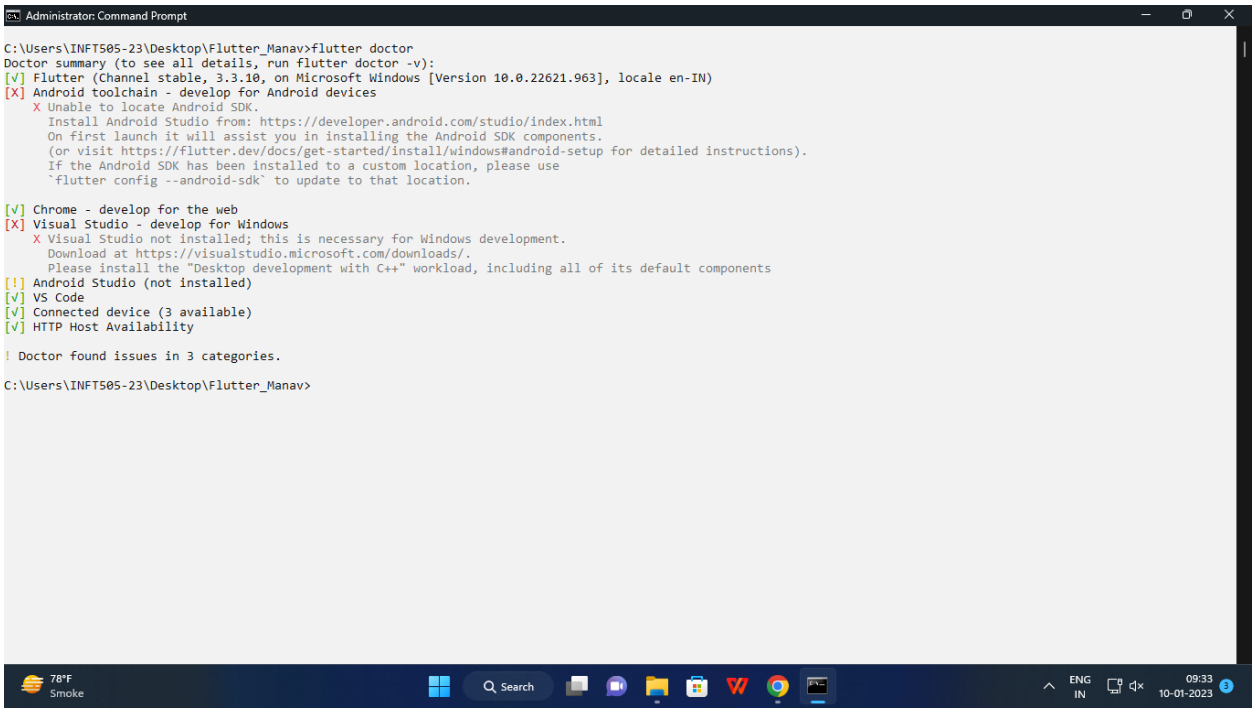


Step 4.3: In the above window, click on New->write path of Flutter bin folder in variable value -> ok -> ok -> ok.

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



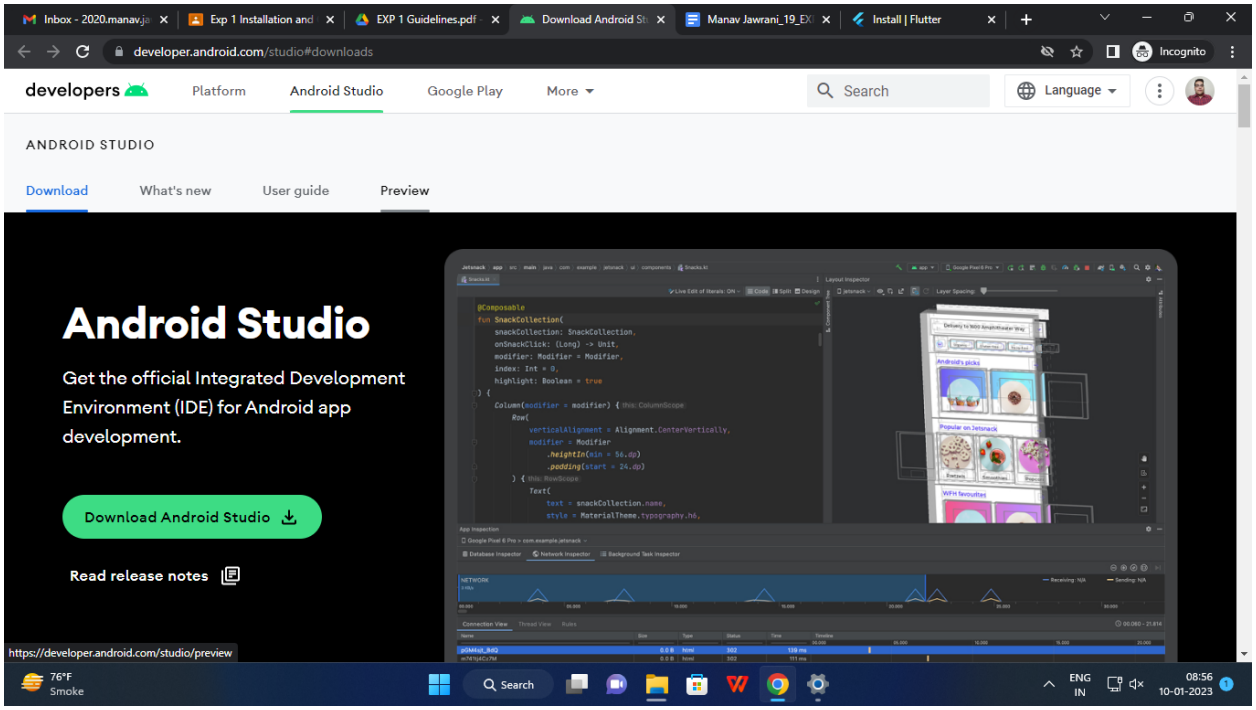
Now, run the **$ flutter doctor** command. This command checks for all the requirements of Flutter app development and displays a report of the status of your Flutter installation.



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.

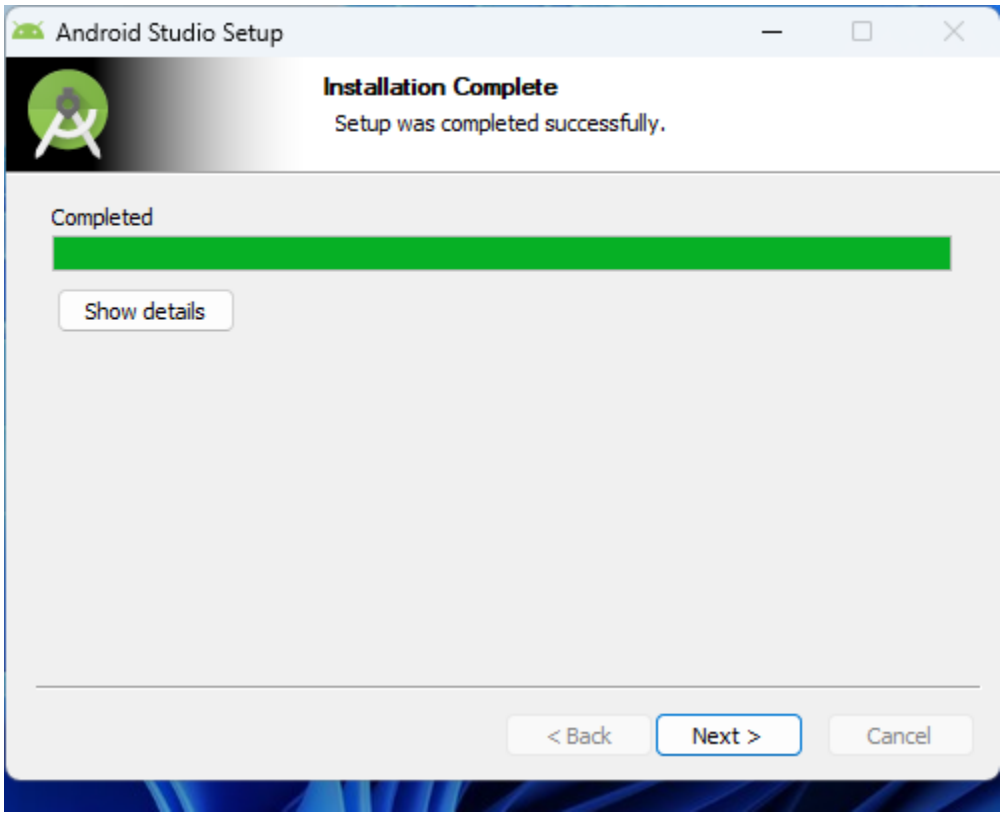
Step 7: Install the Android SDK. If the flutter doctor command does not find the Android SDK tool in your system, then you need first to install the Android Studio IDE. To install Android Studio IDE, do the following steps.

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

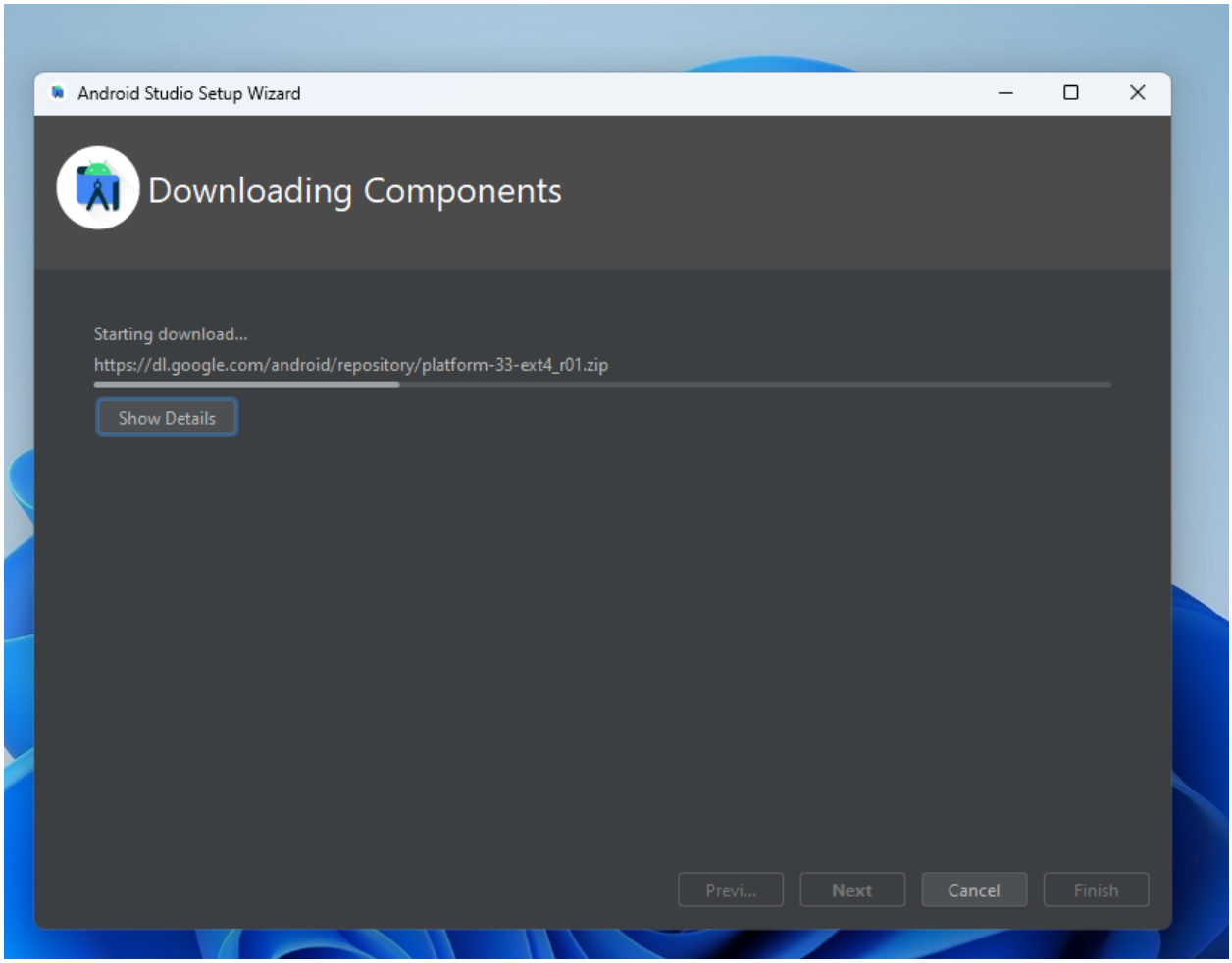


Step 7.2: When the download is complete, open the .exe file and run it. You will get the following dialog box.

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.

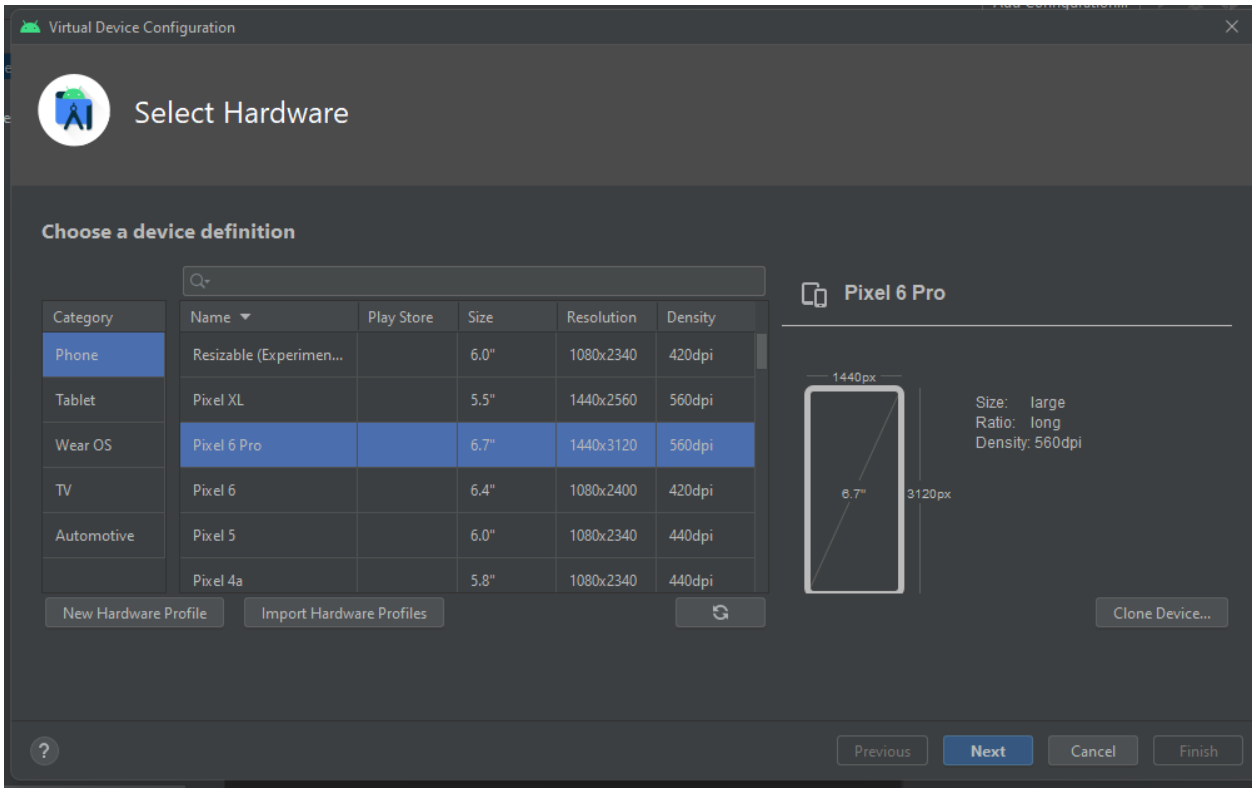


Step 7.5: Run the **$ flutter doctor** command and Run **flutter doctor --android-licenses** command.

```
Administrator: Command Prompt

C:\Users\INFT505-23\Desktop\Flutter_Manav>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, 3.3.10, on Microsoft Windows [Version 10.0.22621.963], locale en-IN)
[√] Android toolchain - develop for Android devices (Android SDK version 33.0.1)
[√] Chrome - develop for the web
[√] Visual Studio - develop for Windows (Visual Studio Community 2022 17.4.3)
[√] Android Studio (version 2021.3)
[√] VS Code
[√] Connected device (3 available)
[√] HTTP Host Availability

• No issues found!

C:\Users\INFT505-23\Desktop\Flutter_Manav>
```

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

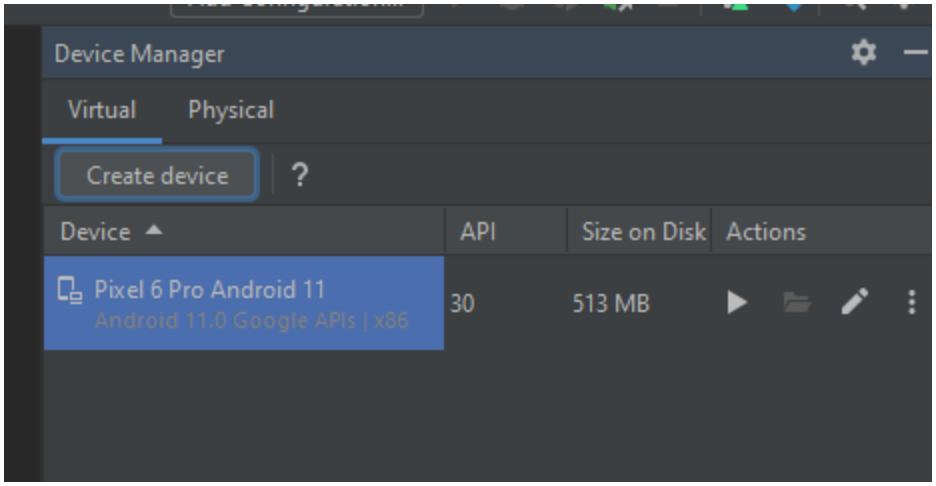
Step 8.1: To set an Android emulator, go to Android Studio > Tools > Android > AVD Manager and select Create Virtual Device. Or, go to Help->Find Action->Type Emulator in the search box. You will get the following screen.



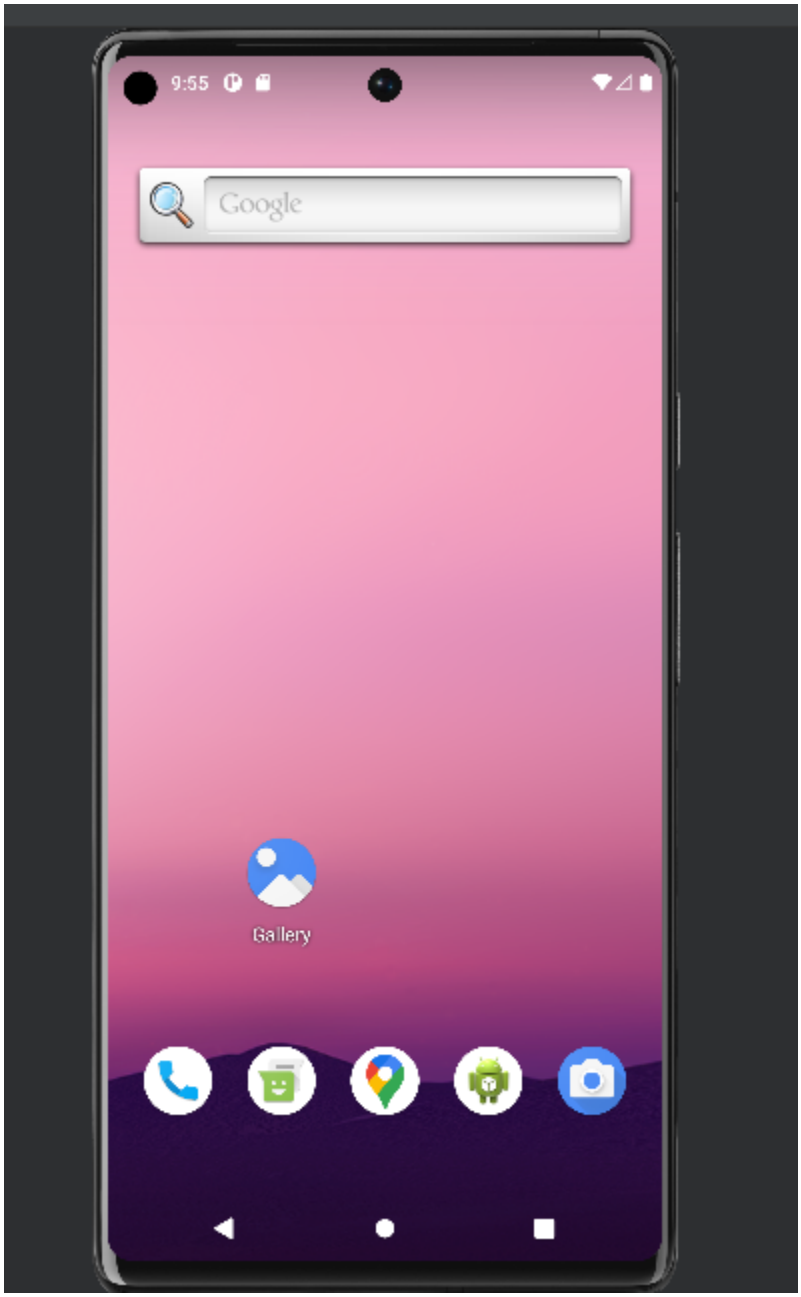
Step 8.2: Choose your device definition and click on Next.

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.


Device Manager
Virtual
Physical
Create device
?
Device
API
Size on Disk
Actions
Pixel 6 Pro Android 11
Android 11.0 Google APIs | x86
30
513 MB

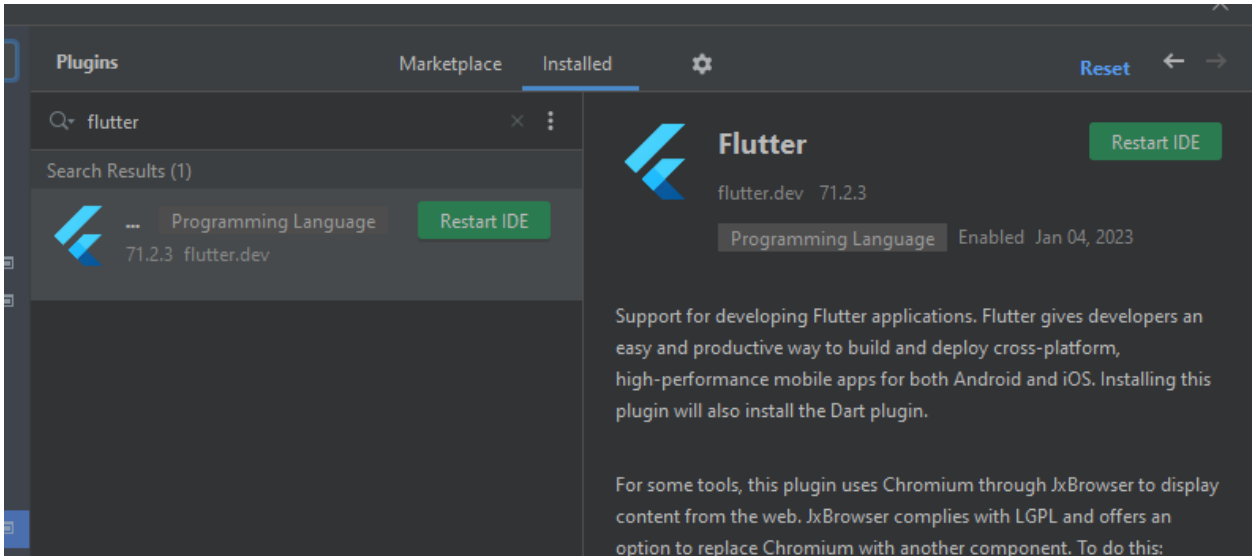
Step 8.5: Last, click on the icon pointed into the red color rectangle. The Android emulator displayed as below screen.



Step 9: Now, install the Flutter and Dart plugin for building Flutter applications 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.

Experiment No:02

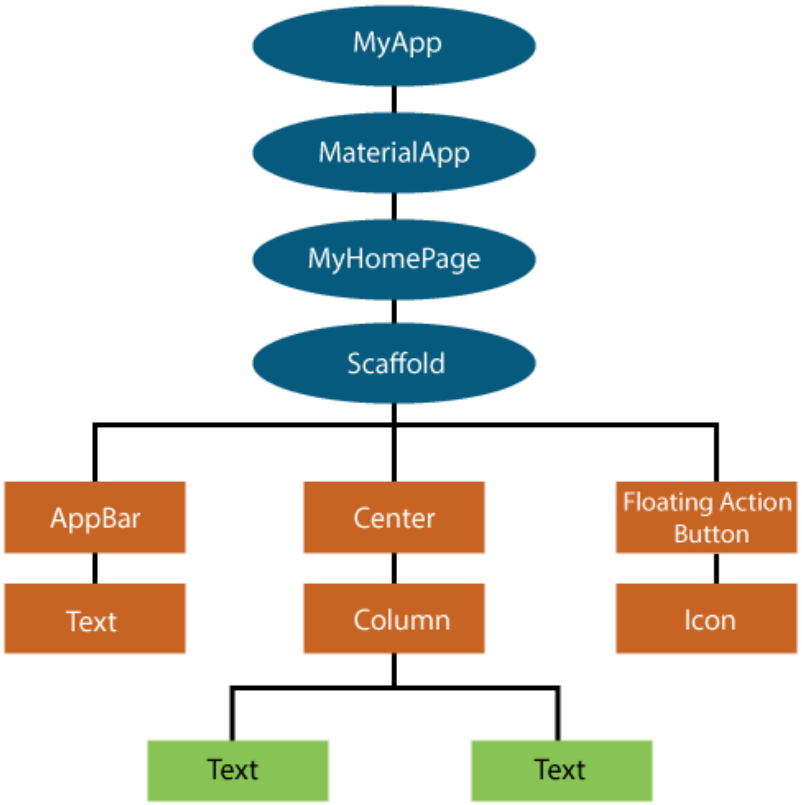
Aim: To design Flutter UI by including common widgets.

Theory:

Whenever you are going to code for building anything in Flutter, it will be inside a widget. The central purpose is to build the app out of widgets. It describes how your app view should look like with their current configuration and state. When you made any alteration in the code, the widget rebuilds its description by calculating the difference of previous and current widget to determine the minimal changes for rendering in UI of the app.

Widgets are nested with each other to build the app. It means the root of your app is itself a widget, and all the way down is a widget also. For example, a widget can display something, can define design, can handle interaction, etc.

The below image is a simple visual representation of the widget tree.



Types of Widget

We can split the Flutter widget into two categories:

1. Visible (Output and Input)
2. Invisible (Layout and Control)

Common Widgets in Flutter

Flutter provides a wide range of widgets that developers can use to build rich and interactive user interfaces. These widgets serve various purposes, from displaying text and images to handling user input and managing layouts.

In the below Flutter code, the following widgets are used:

1. MaterialApp: Represents the root widget of the application and configures the overall theme and home page.

2. Scaffold: Provides a basic layout structure for the home page, including an app bar and body content area.

3. AppBar: Displays a toolbar at the top of the screen with a title.

4. Center: Centers its child widget both vertically and horizontally within its container.

5. Column: Arranges its children widgets vertically.

6. Text: Displays text on the screen with customizable styles.

7. SizedBox: Provides a box with a specified size, used for adding space between widgets.

8. TextField: Allows users to input text.

9. ElevatedButton: Represents a button with a raised appearance.

10. TextButton: Represents a button with text but no background color or elevation.

11. IconButton: Represents a button with an icon.

12. Icon: Displays an icon image.

13. Image.asset: Displays an image loaded from the assets folder of the project.

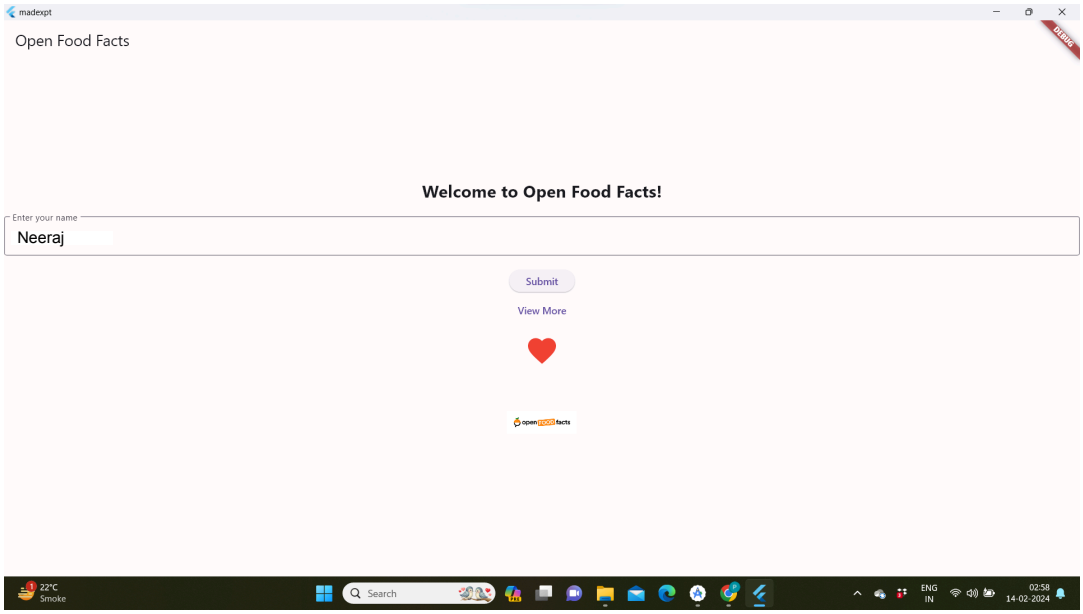
These widgets are used to create a basic UI with text, input fields, buttons, and images.

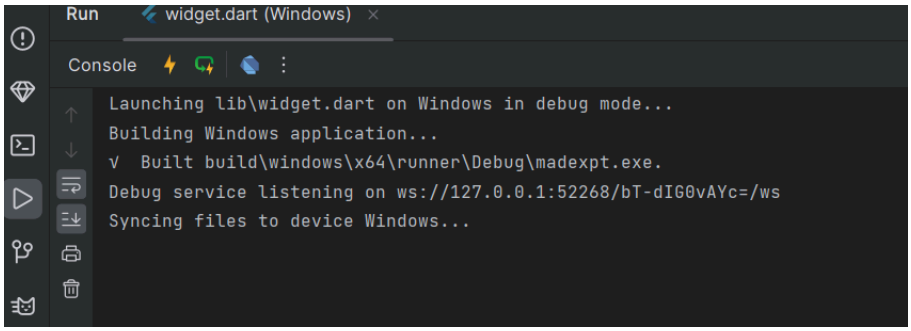
Code:

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

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

class OpenFoodFactsApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Open Food Facts',
      home: Scaffold(
        appBar: AppBar(
          title: Text('Open Food Facts'),
          backgroundColor: Colors.purple.shade100,
        ),
        body: Center(
          child: SingleChildScrollView(
            padding: EdgeInsets.all(20.0),
            child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              children: <Widget>[
                Text(
                  'Welcome to Open Food Facts!',
                  style: TextStyle(
                    fontSize: 20.0,
                    fontWeight: FontWeight.bold,
                  ),
                  textAlign: TextAlign.center,
                ),
                SizedBox(height: 20.0),
                TextField(
                  decoration: InputDecoration(
                    labelText: 'Enter your name',
                    border: OutlineInputBorder(),
                  ),
                ),
                SizedBox(height: 20.0),
                ElevatedButton(
                  onPressed: () {
                    // Submit logic
                  },
                  style: ElevatedButton.styleFrom(
```

```
                primary: Colors.purple.shade100,
              ),
              child: Text('Submit'),
            ),
            SizedBox(height: 10.0),
            TextButton(
              onPressed: () {
                // View more logic
              },
              child: Text(
                'View More',
                style: TextStyle(
                  color: Colors.deepPurple,
                ),
              ),
            ),
            SizedBox(height: 20.0),
            Icon(
              Icons.favorite,
              color: Colors.red,
              size: 40.0,
            ),
            SizedBox(height: 20.0),
            Image.asset(
              'images/openfoodfactslogo.png', // make sure the image is present in your assets
              height: 40.0,
            ),
          ],
        ),
      ),
    ),
  );
 }
}
```

Output:





Conclusion:

In conclusion, designing Flutter UIs with common widgets provides a robust foundation for creating beautiful, functional, and responsive applications. By leveraging the versatility and flexibility of these widgets, developers can efficiently build UIs that deliver an exceptional user experience on both iOS and Android platforms.

Experiment No:03

Aim : To explore Flutter Widgets like image,icon and to use custom fonts.

Theory :

In the below code we use the following widgets:

1.MaterialApp: This widget represents a Flutter application that uses material design.
2.Scaffold: Scaffold is a layout structure widget from the Material library that provides a default layout structure for the app. It includes an app bar, a body, and other structural elements.
3.AppBar: AppBar is a Material Design app bar that displays the title and other actions above the app's main content.
4.SingleChildScrollView: This widget enables scrolling when the content is too large to fit within the visible area. It allows the child widget to be scrolled in one direction.
5.Column: Column is a layout widget that arranges its children vertically, one after another.
6.Image: The Image widget displays an image. In this code, it's used to display the login page's logo.
7.SizedBox: SizedBox is a widget that creates a fixed-size box. It's used here to create space between widgets vertically.
8.TextField: TextField is a widget that allows users to enter text. It's used here for the email and password input fields.
9.ElevatedButton: ElevatedButton is a button widget with a raised appearance, typically used for primary actions.
10.TextButton: TextButton is a button widget with only text, suitable for secondary actions.
11.Text: Text is a widget that displays a string of text. It's used to provide labels and button text in this code.

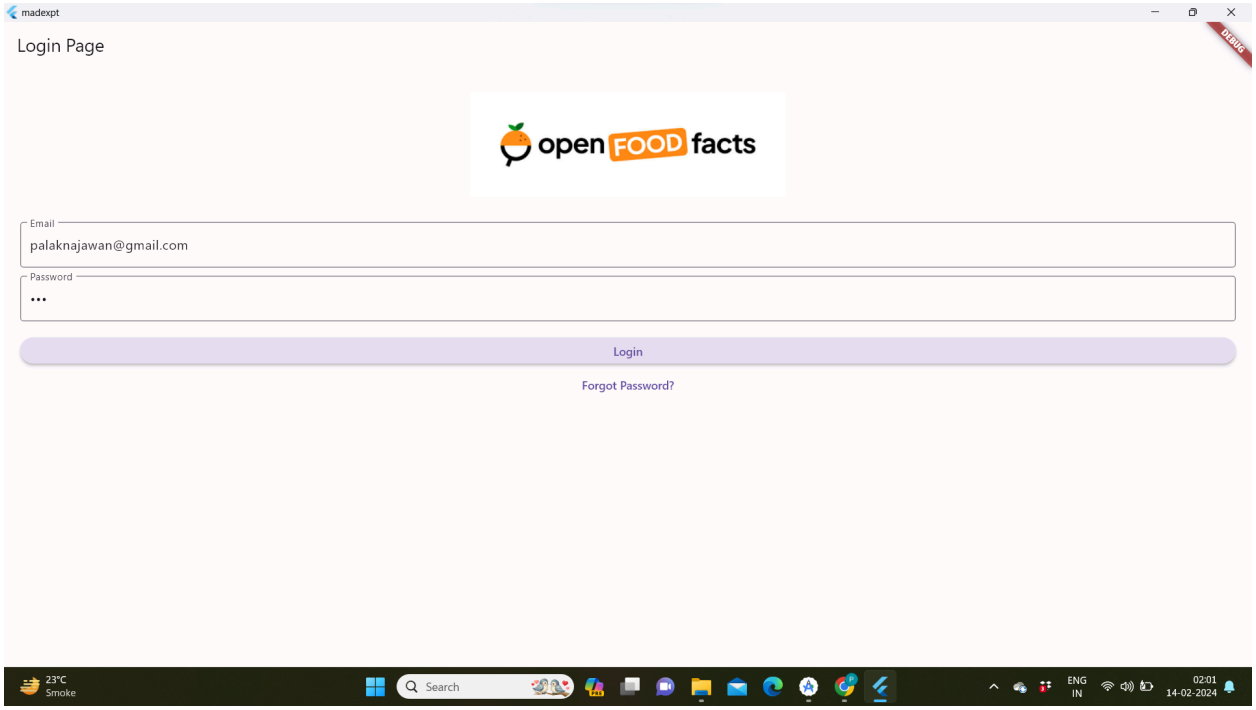
These widgets are used to create a simple login page with email and password input fields, along with login and forgot password buttons.

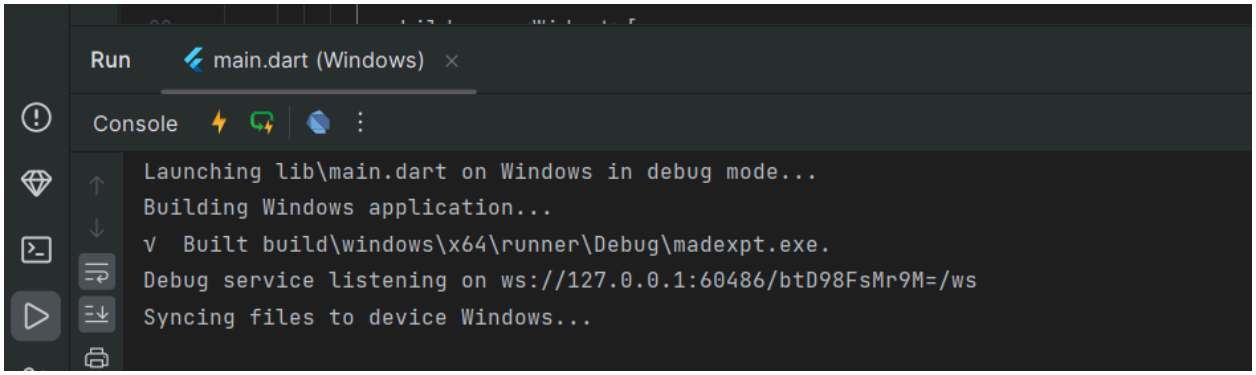
Code :

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

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

class LoginPage extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
  return MaterialApp(
    home: Scaffold(
     appBar: AppBar(
       title: Text('Login Page'),
     ),
     body: SingleChildScrollView(
       padding: EdgeInsets.all(20.0),
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         crossAxisAlignment: CrossAxisAlignment.stretch,
         children: <Widget>[
           Image(
             image: AssetImage('images/download.png'),
             height: 150.0,
             width: 150.0,
           ),
           SizedBox(height: 20.0),
           TextField(
             decoration: InputDecoration(
               labelText: 'Email',
               border: OutlineInputBorder(),
             ),
           ),
           SizedBox(height: 10.0),
           TextField(
```

```
            obscureText: true,
            decoration: InputDecoration(
              labelText: 'Password',
              border: OutlineInputBorder(),
            ),
          ),
          SizedBox(height: 20.0),
          ElevatedButton(
            onPressed: () {
              // Add login logic here
            },
            child: Text('Login'),
          ),
          SizedBox(height: 10.0),
          TextButton(
            onPressed: () {
              // Add forgot password logic here
            },
            child: Text('Forgot Password?'),
          ),
        ],
      ),
    ),
  ),
 );
}
}
```

Output:





Conclusion:

Overall, mastering these widgets empowers developers to create visually appealing and cohesive user interfaces that align with design requirements and brand aesthetics. Additionally, Flutter's flexibility and ease of use make it straightforward to incorporate images, icons, and custom fonts into applications, enhancing the overall user experience and engagement.

Experiment No. : 04

Aim: To create an interactive Form using form widget

Theory:

Flutter Forms

Forms are an integral part of all modern mobile and web applications. It is mainly used to interact with the app as well as gather information from the users. They can perform many tasks, which depend on the nature of your business requirements and logic, such as authentication of the user, adding user, searching, filtering, ordering, booking, etc. A form can contain text fields, buttons, checkboxes, radio buttons, etc.

Creating Form

Flutter provides a Form widget to create a form. The form widget acts as a container, which allows us to group and validate the multiple form fields. When you create a form, it is necessary to provide the GlobalKey. This key uniquely identifies the form and allows you to do any validation in the form fields.
The form widget uses child widget TextFormField to provide the users to enter the text field. This widget renders a material design text field and also allows us to display validation errors when they occur.

Form validation

Validation is a method, which allows us to correct or confirms a certain standard. It ensures the authentication of the entered data.
Validating forms is a common practice in all digital interactions. To validate a form in a flutter, we need to implement mainly three steps.

Now let's go through the key components and features used in the code below:
1.Material Design: The app follows the Material Design principles provided by the Flutter framework, ensuring a consistent and visually appealing user interface.
2.StatefulWidget: The SignUpForm class is a stateful widget, allowing it to maintain state (such as the user input) and update the UI accordingly.

3.Form Widget: The Form widget is used to group the form fields together. It allows us to perform form validation and submission easily.
4.GlobalKey: The GlobalKey<FormState> is used to uniquely identify the form widget. This key is necessary for performing operations like form validation and saving.
5.TextFormField: These widgets are used for user input. EachTextFormField represents a field in the form, such as username, email, and password. They include properties like decoration for styling and validator for input validation.
6.Padding: The Padding widget adds padding around its child widget, ensuring proper spacing between UI elements.
7.SizedBox: These widgets are used to add empty space (vertical height in this case) between UI elements. They help improve the layout and readability of the form.
8.ElevatedButton: This button widget triggers the sign-up action when pressed. It is styled as an elevated button, following the Material Design guidelines.
9.AlertDialog: When the sign-up process is complete, an AlertDialog is shown to inform the user about the successful sign-up. It includes a title, content, and an "OK" button to close the dialog.
10.showDialog: The showDialog function is used to display the AlertDialog on the screen. It takes the BuildContext and a builder function that returns the dialog widget.
11.Image.asset: This widget is used to display the logo image at the top of the sign-up form. It loads the image from the assets directory using the provided image path.
12.TextButton: This button widget is used inside the AlertDialog for dismissing the dialog when pressed.
Overall, these components work together to create an interactive sign-up form with input validation and a user-friendly interface.

Code:

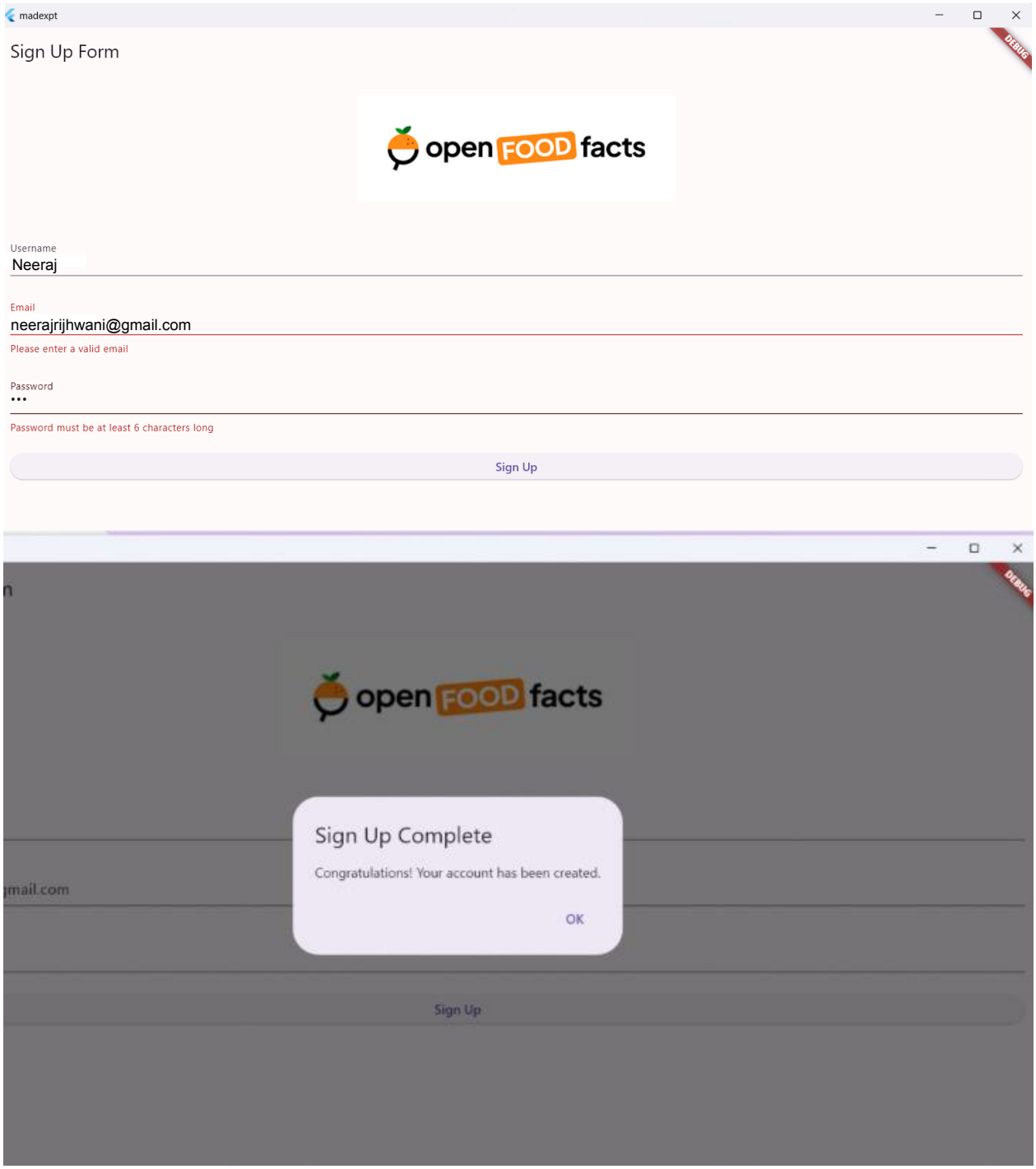
```
import 'package:flutter/material.dart';

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

class MyApp extends StatelessWidget {
```

```
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Sign Up Form',
     theme: ThemeData(
       primarySwatch: Colors.blue,
       visualDensity: VisualDensity.adaptivePlatformDensity,
     ),
     home: SignUpForm(),
   );
 }
}

class SignUpForm extends StatefulWidget {
 @override
 _SignUpFormState createState() => _SignUpFormState();
}

class _SignUpFormState extends State<SignUpForm> {
 final GlobalKey<FormState> _formKey = GlobalKey<FormState>();
 TextEditingController _usernameController = TextEditingController();
 TextEditingController _emailController = TextEditingController();
 TextEditingController _passwordController = TextEditingController();

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text('Sign Up Form'),
     ),
     body: Padding(
       padding: const EdgeInsets.all(16.0),
       child: Form(
         key: _formKey,
         child: Column(
           crossAxisAlignment: CrossAxisAlignment.stretch,
           children: <Widget>[
             Image(
               image: AssetImage('images/download.png'),
               height: 150.0,
               width: 150.0,
             ),
             SizedBox(height: 24.0),
             TextFormField(
               controller: _usernameController,
               decoration: InputDecoration(labelText: 'Username'),
               validator: (value) {
                 if (value == null || value.isEmpty) {
                   return 'Please enter your username';
```

```
                }
                return null;
              },
            ),
            SizedBox(height: 16.0),
            TextFormField(
              controller: _emailController,
              decoration: InputDecoration(labelText: 'Email'),
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'Please enter your email';
                }
                if (!value.contains('@')) {
                  return 'Please enter a valid email';
                }
                return null;
              },
            ),
            SizedBox(height: 16.0),
            TextFormField(
              controller: _passwordController,
              decoration: InputDecoration(labelText: 'Password'),
              obscureText: true,
              validator: (value) {
                if (value == null || value.isEmpty) {
                  return 'Please enter your password';
                }
                if (value.length < 6) {
                  return 'Password must be at least 6 characters long';
                }
                return null;
              },
            ),
            SizedBox(height: 24.0),
            ElevatedButton(
              onPressed: () {
                if (_formKey.currentState!.validate()) {
                  _showSignUpCompleteDialog();
                }
              },
              child: Text('Sign Up'),
            ),
          ],
        ),
      ),
    ),
  );
}
```

```
void _showSignUpCompleteDialog() {
  showDialog(
    context: context,
    builder: (BuildContext context) {
      return AlertDialog(
        title: Text('Sign Up Complete'),
        content: Text('Congratulations! Your account has been created.'),
        actions: <Widget>[
          TextButton(
            onPressed: () {
              Navigator.of(context).pop();
            },
            child: Text('OK'),
          ),
        ],
      );
    },
  );
}

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

Output:



Conclusion:

In conclusion, creating an interactive form using the form widget in Flutter enables developers to build robust, user-friendly applications that efficiently collect and process user input. This approach enhances usability, improves data accuracy, and contributes to a positive user experience.

Experiment 5

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

Theory :

Navigation and routing are some of the core concepts of all mobile applications, which allows the user to move between different pages. We know that every mobile application contains several screens for displaying different types of information. **For example,** an app can have a screen that contains various products. When the user taps on that product, immediately it will display detailed information about that product.

In Flutter, the screens and pages are known as **routes,** and these routes are just a widget. In Android, a route is similar to an **Activity,** whereas, in iOS, it is equivalent to a **ViewController.**

In any mobile app, navigating to different pages defines the workflow of the application, and the way to handle the navigation is known as **routing.** Flutter provides a basic routing class **MaterialPageRoute** and two methods **Navigator.push()** and **Navigator.pop()** that shows how to navigate between two routes. The following steps are required to start navigation in your application.

Step 1: First, you need to create two routes.

Step 2: Then, navigate to one route from another route by using the Navigator.push() method.

Step 3: Finally, navigate to the first route by using the Navigator.pop() method.

Let us take a simple example to understand the navigation between two routes:

Code:

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

void main() {
 runApp(MaterialApp(
   initialRoute: '/',
   routes: {
     '/': (context) => MobileLoginPage(),
     '/form': (context) => SignUpPage(),
```

```
  },
 ));
}

class MobileLoginPage extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text('Login'),
     ),
     body: Padding(
       padding: const EdgeInsets.all(20.0),
       child: Column(
         mainAxisAlignment: MainAxisAlignment.center,
         children: [
           Image(
             image: AssetImage('images/download.png'),
             height: 150.0,
             width: 150.0,
           ),
           SizedBox(height: 20),
           TextFormField(
             decoration: InputDecoration(
               labelText: 'Email',
               border: OutlineInputBorder(),
             ),
           ),
           SizedBox(height: 20),
           TextFormField(
             decoration: InputDecoration(
               labelText: 'Password',
               border: OutlineInputBorder(),
             ),
             obscureText: true,
           ),
           SizedBox(height: 20),
           ElevatedButton(
             onPressed: () {
               // Add login logic here
             },
             child: Text('Login'),
           ),
           SizedBox(height: 10),
           Row(
             mainAxisAlignment: MainAxisAlignment.center,
             children: [
               Text("Don't have an account? "),
               TextButton(
```

```
                onPressed: () {
                  Navigator.pushNamed(context, '/form');
                },
                child: Text('Sign Up'),
              ),
            ],
          ),
        ],
      ),
    ),
  );
 }
}

class SignUpPage extends StatefulWidget {
 @override
 _SignUpPageState createState() => _SignUpPageState();
}

class _SignUpPageState extends State<SignUpPage> {
 final _formKey = GlobalKey<FormState>();
 String? _username;
 String? _email;
 String? _phoneNumber;
 String? _password;
 String? _foodPreference;
 List<String> _foodPreferences = [
   'Vegetarian',
   'Non Vegetarian',
   'Vegan',
 ];

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text('Sign Up Page'),
     ),
     body: Padding(
       padding: EdgeInsets.all(16.0),
       child: Form(
         key: _formKey,
         child: SingleChildScrollView(
           child: Column(
             crossAxisAlignment: CrossAxisAlignment.stretch,
             children: <Widget>[
               Image(
                 image: AssetImage('images/download.png'),
                 height: 150.0,
```

```
                width: 150.0,
              ),
              TextFormField(
                decoration: InputDecoration(labelText: 'Username'),
                validator: (value) {
                  if (value == null || value.isEmpty) {
                    return 'Please enter a username';
                  }
                  return null;
                },
                onSaved: (value) => _username = value,
              ),
              TextFormField(
                decoration: InputDecoration(labelText: 'Email'),
                validator: (value) {
                  if (value == null || value.isEmpty) {
                    return 'Please enter an email address';
                  }
                  return null;
                },
                onSaved: (value) => _email = value,
              ),
              TextFormField(
                decoration: InputDecoration(labelText: 'Phone Number'),
                validator: (value) {
                  if (value == null || value.isEmpty) {
                    return 'Please enter a phone number';
                  }
                  return null;
                },
                onSaved: (value) => _phoneNumber = value,
              ),
              TextFormField(
                decoration: InputDecoration(labelText: 'Password'),
                obscureText: true,
                validator: (value) {
                  if (value == null || value.isEmpty) {
                    return 'Please enter a password';
                  }
                  return null;
                },
                onSaved: (value) => _password = value,
              ),
              DropdownButtonFormField<String>(
                value: _foodPreference,
                items: _foodPreferences.map((preference) {
                  return DropdownMenuItem(
                    value: preference,
                    child: Text(preference),
```

```
                  );
                }).toList(),
                onChanged: (value) {
                  setState(() {
                    _foodPreference = value;
                  });
                },
                decoration: InputDecoration(labelText: 'Food Preference'),
                validator: (value) {
                  if (value == null) {
                    return 'Please select a food preference';
                  }
                  return null;
                },
              ),
              SizedBox(height: 20),
              ElevatedButton(
                onPressed: () {
                  if (_formKey.currentState!.validate()) {
                    _formKey.currentState!.save();
                    // After successful form submission, navigate back to
login page
                    Navigator.pop(context);
                  }
                },
                child: Text('Submit'),
              ),
            ],
          ),
        ),
      ),
    );
  }
}
```

Explanation of the code:

This code is a Flutter application that implements two screens: a login page and a sign-up page. Here's an explanation of the code:

1. Main Function and MaterialApp:

- The **main()** function is the entry point of the application. It calls **runApp()** with a **MaterialApp** widget.
- **MaterialApp** is a convenience widget that provides several features commonly required in mobile applications, such as navigation and theme management.

- **initialRoute** specifies the initial route of the application, which is '/' (the login page).
- **routes** defines the named routes of the application, mapping route names to corresponding builder functions.

2. Login Page (MobileLoginPage):
- This page displays a login form with fields for email and password.
- It uses a **Scaffold** widget to provide the basic structure of the page, including an AppBar with the title "Login".
- The form fields are wrapped in a **Column** widget to arrange them vertically.
- A **TextFormField** widget is used for email and password input.
- An **ElevatedButton** widget is used for the login button, which currently has no functionality.
- A **TextButton** widget is used to navigate to the sign-up page when clicked.

3. **Sign-Up Page (SignUpPage)**:
- This page allows users to sign up with a username, email, phone number, password, and food preference.
- It also uses a **Scaffold** widget for the basic page structure.
- The form fields are wrapped in a **Form** widget, allowing for form validation and submission.
- Each form field is represented by a **TextFormField** widget with appropriate decoration and validation logic.
- The food preference field uses a **DropdownButtonFormField** widget to display a dropdown menu of options.
- An **ElevatedButton** widget is used to submit the form. Upon successful validation, the form data is saved and the sign-up page is closed (popped from the navigation stack).

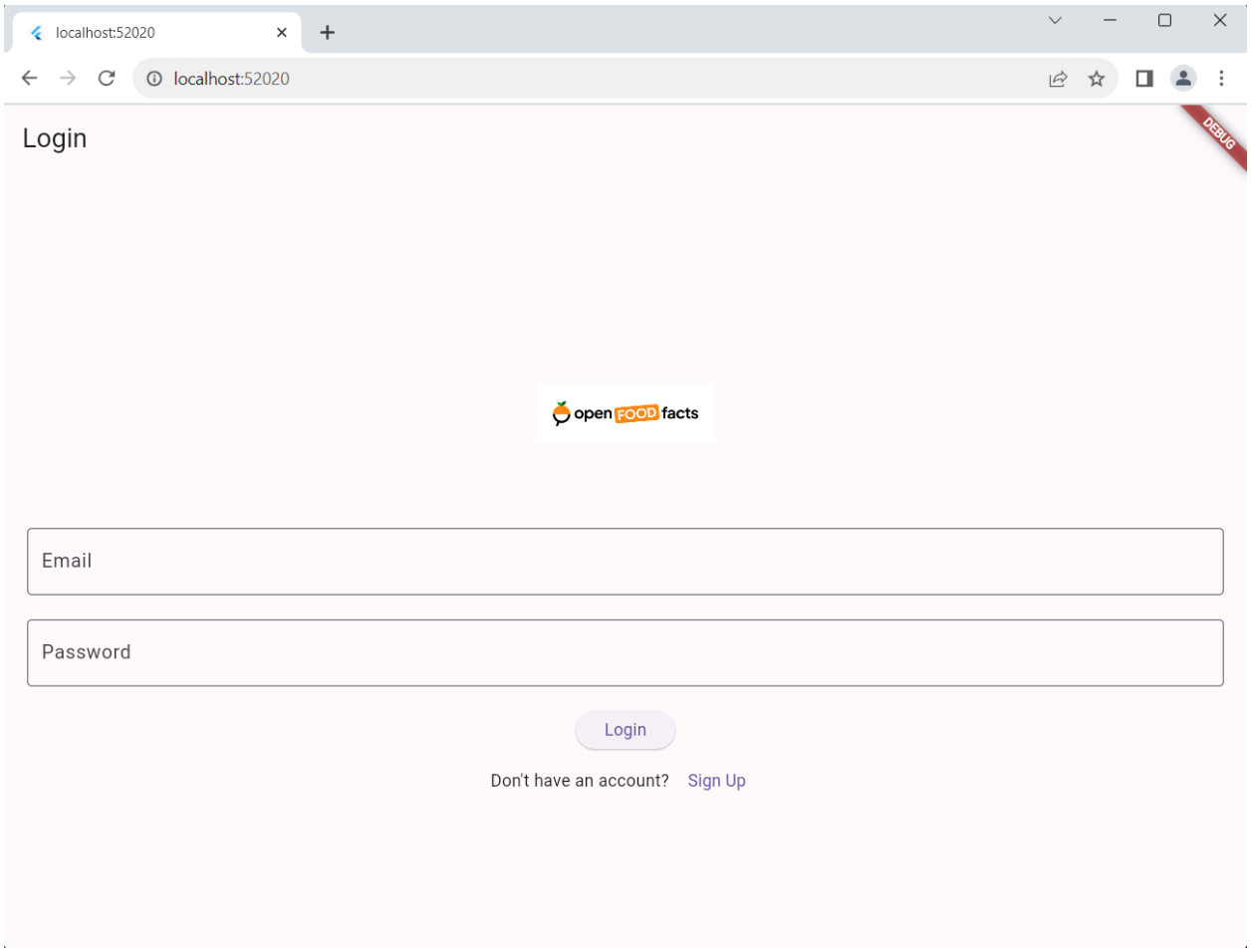
4. State Management:
- The sign-up page (**SignUpPage**) is a stateful widget (**StatefulWidget**) because it needs to maintain the form state.
- It uses a GlobalKey<FormState> to access the state of the form for validation and saving.

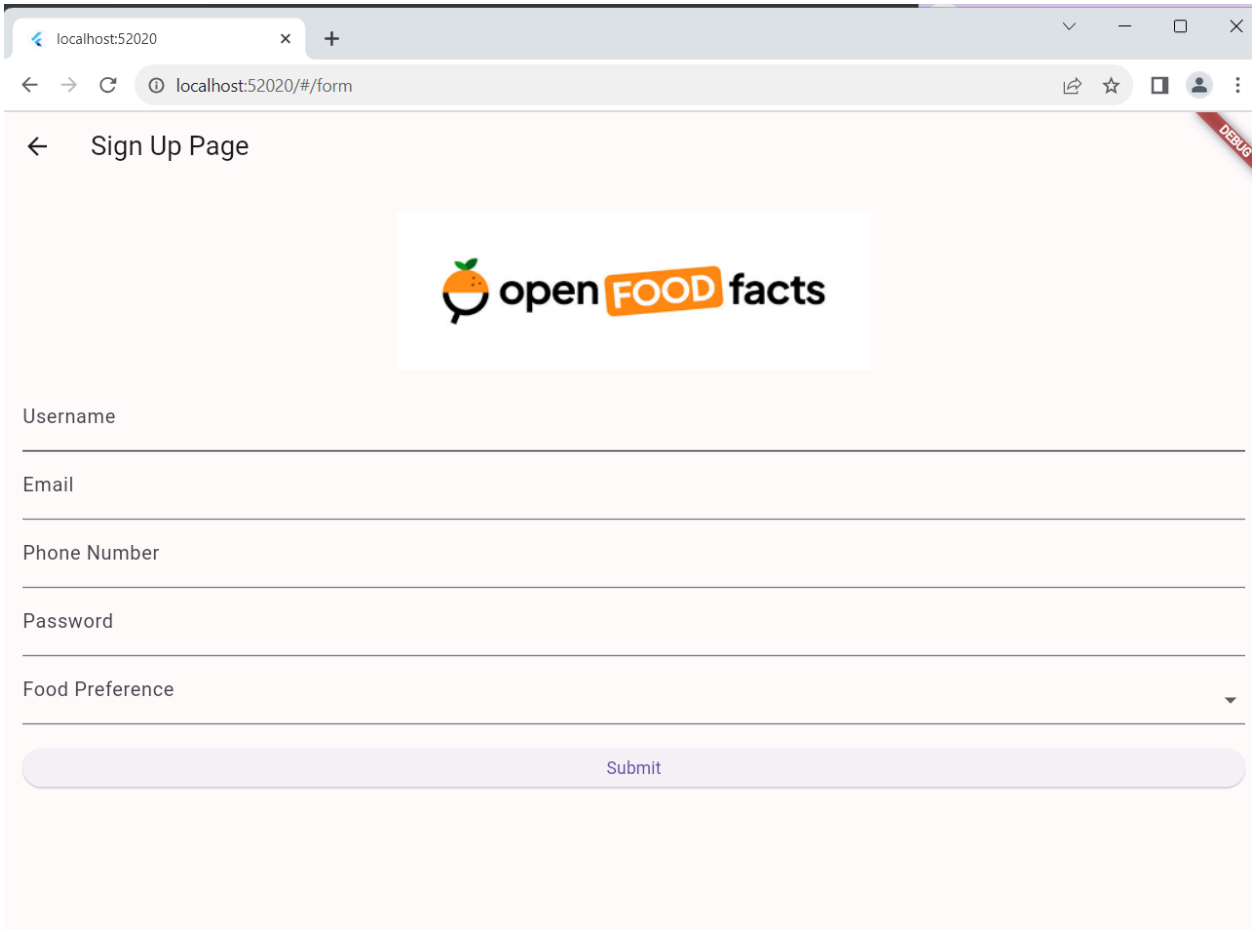
5. Navigation:

- Navigation between the login page and sign-up page is handled using named routes. When the "Sign Up" button is pressed on the login page, it navigates to the sign-up page ('/form'). After successful sign-up, the sign-up page is closed and the user is taken back to the login page.

Overall, this code provides a basic login and sign-up functionality for a mobile application built with Flutter. It demonstrates how to create forms, handle form submission, perform form validation, and navigate between screens using named routes.

Output:





Conclusion:

In this practical, we developed a Flutter application featuring login and sign-up functionalities. Leveraging named routes, we enabled smooth navigation between the login and sign-up pages, ensuring a structured approach to screen transitions. Using `TextFormField`, we created input fields for email, password, username, phone number, and food preference, implementing validation to maintain data integrity. Through `GlobalKey<FormState>`, we managed form state, facilitating validation and submission processes. The user interface was designed with visually appealing elements such as `Scaffold`, `AppBar`, and images, enhancing the overall user experience.

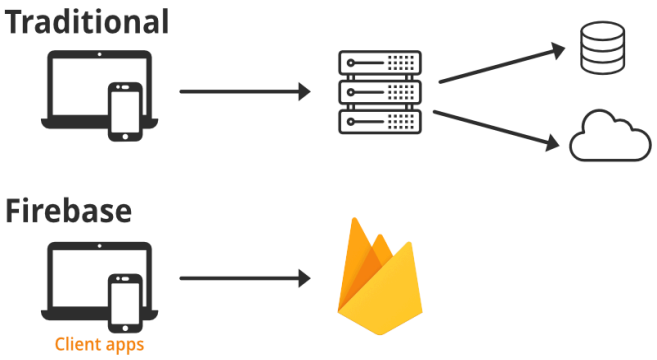
Experiment-6

Aim: To Connect Flutter UI with firebase database.

Theory:

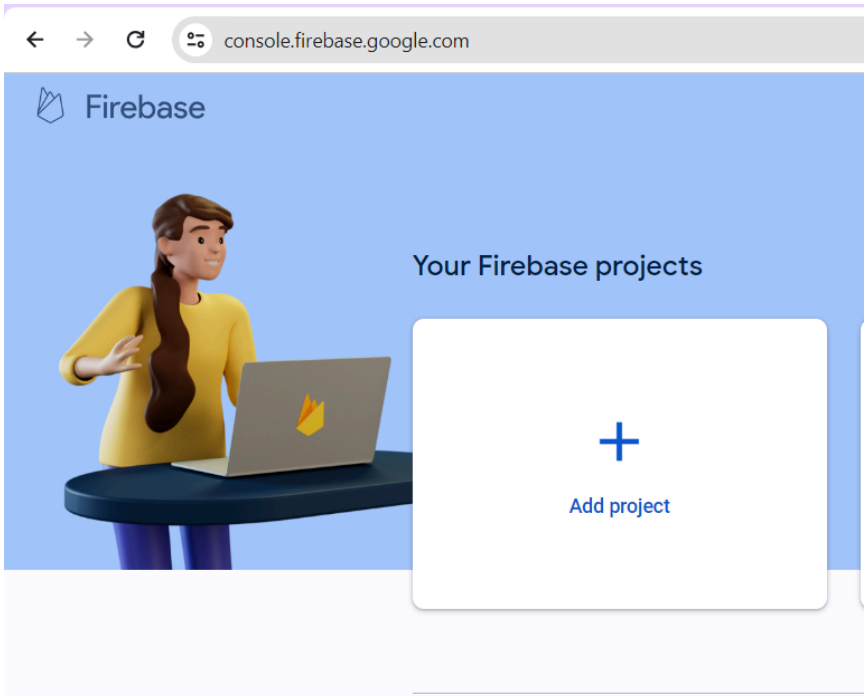
Firebase Authentication is a preconfigured backend service that makes it really easy to integrate with a mobile app using an SDK. You don't have to maintain any backend infrastructure for the authentication process and Firebase supports integration with popular identity providers such as Google, Facebook, and GitHub.

Firebase is a product of Google which helps developers to build, manage, and grow their apps easily. It helps developers to build their apps faster and in a more secure way. No programming is required on the firebase side which makes it easy to use its features more efficiently. It provides services to android, ios, web, and unity. It provides cloud storage. It uses NoSQL for the database for the storage of data.

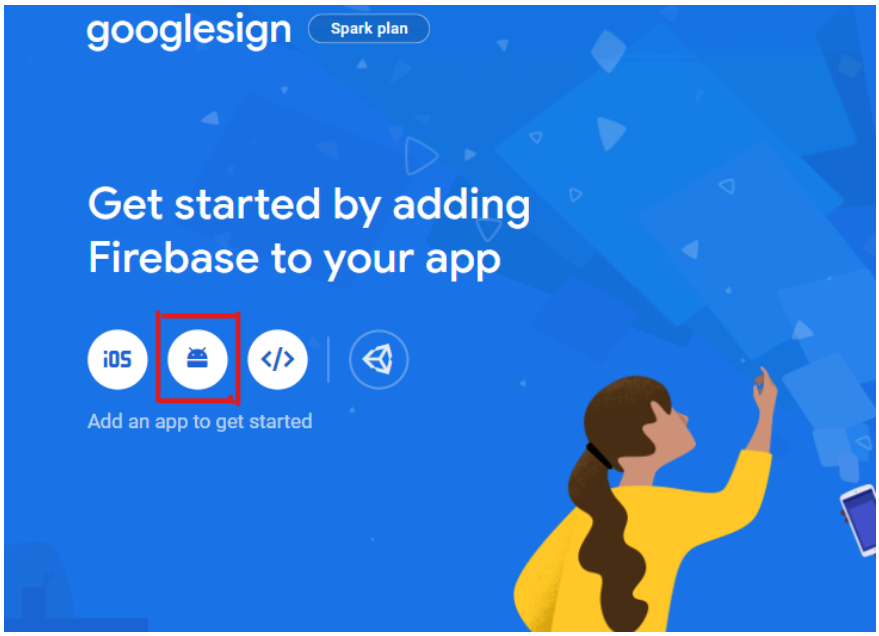


Step by Step Implementation

Step 1: First, you have to visit the Firebase console. Now let's move to the next step. Click on the "Add project" as shown in the below image.

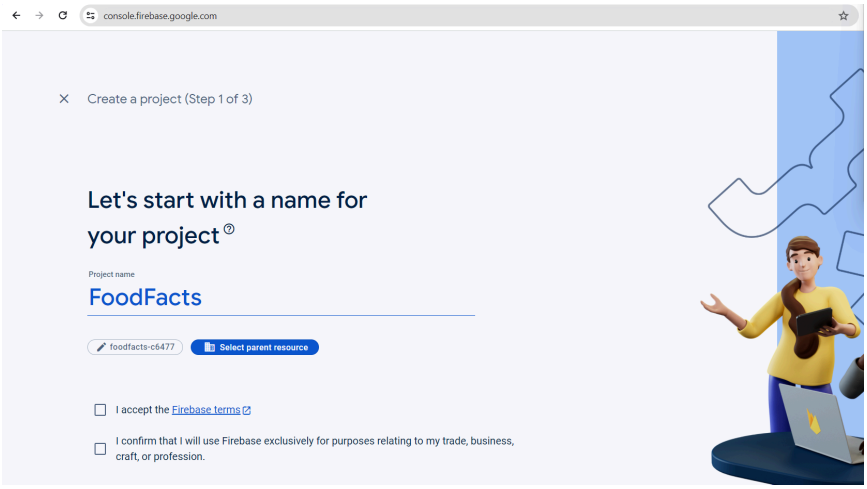


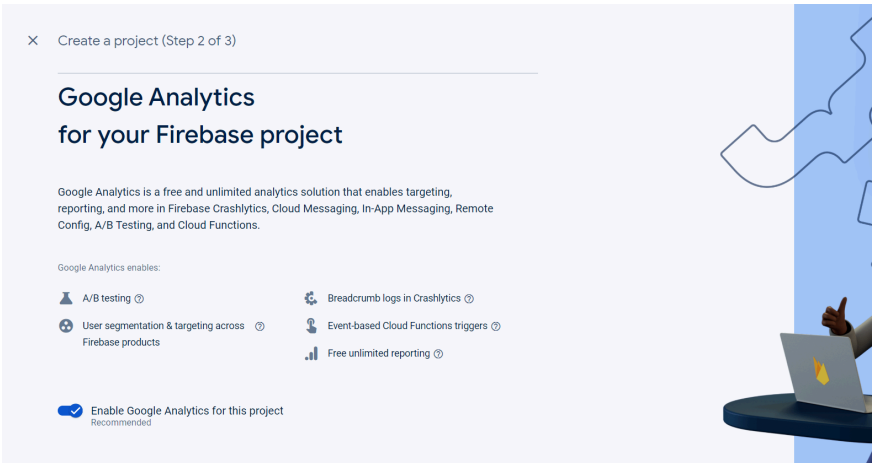
Step 2: Provide the firebase project name. Let's give it "geeksforgeeks". Then click on "Continue". Disable the "Enable Google Analytics for this project" because we don't need this and click on "Create project". It takes some time to wait till the project is created, After that click on "Continue". Now there will be a screen, You to find the Android button and click on it as shown in the below image.

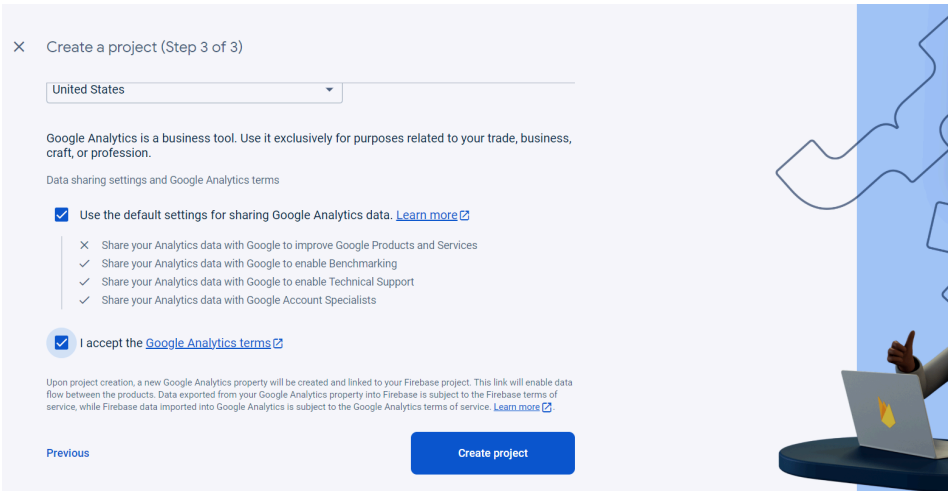


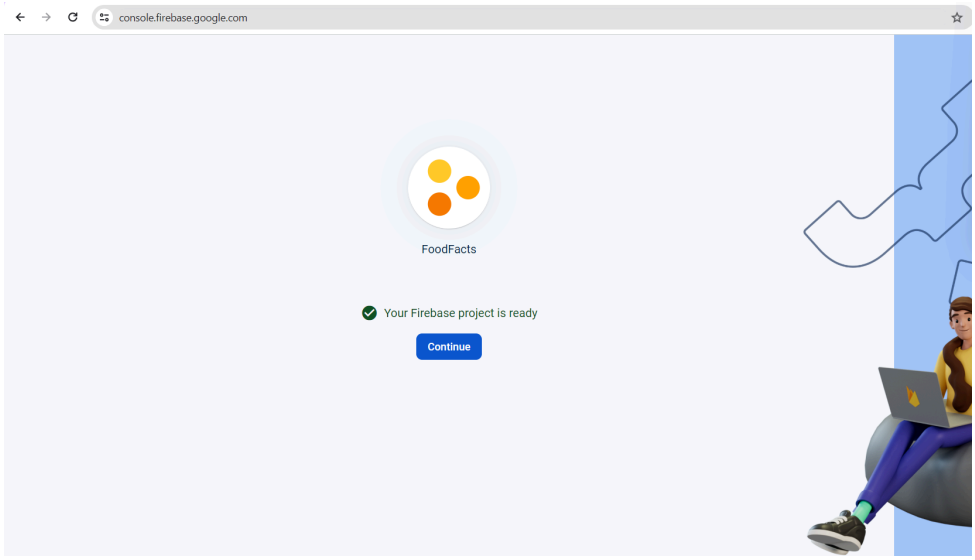
Now time to add firebase to your Android App.

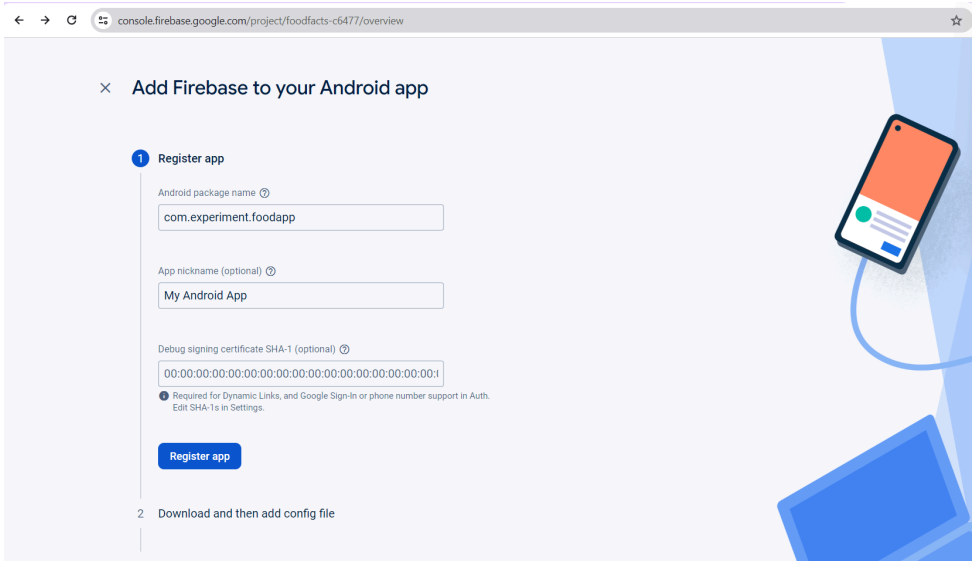
Step 3: We have to give the “Android package name”, where it can be found????. Don’t know, Let’s find this in the next step.

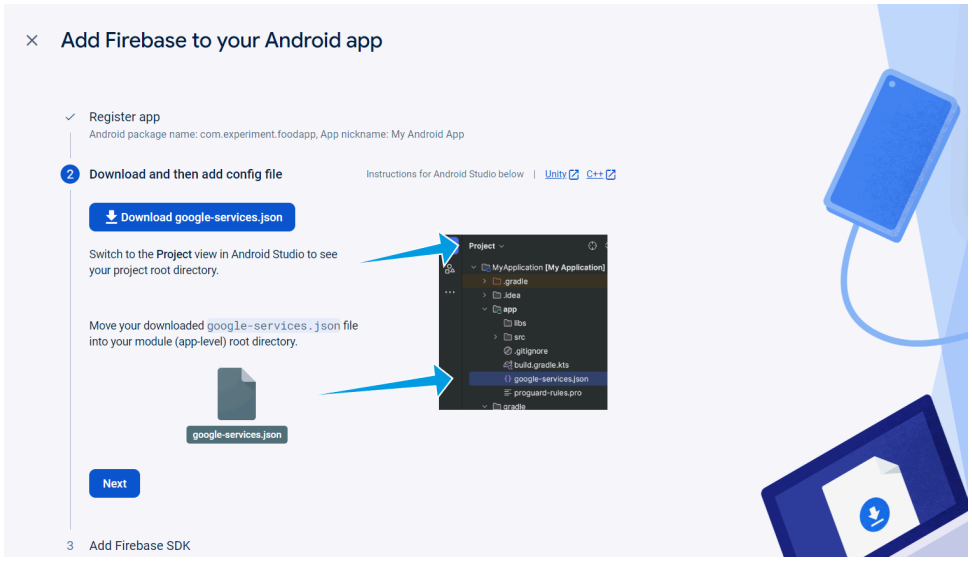






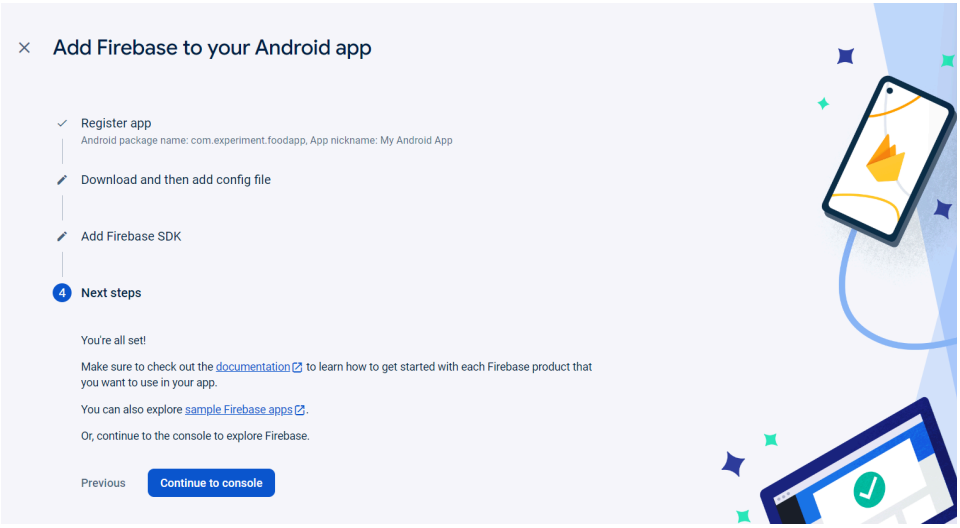

console.firebase.google.com
FoodFacts
Your Firebase project is ready
Continue


console.firebase.google.com/project/foodfacts-c6477/overview
Add Firebase to your Android app
1 Register app
Android package name
com.experiment.foodapp
App nickname (optional)
My Android App
Debug signing certificate SHA-1 (optional)
Required for Dynamic Links, and Google Sign-In or phone number support in Auth.
Edit SHA-1s in Settings.
Register app
2 Download and then add config file


Add Firebase to your Android app
Register app
Android package name: com.experiment.foodapp, App nickname: My Android App
2 Download and then add config file
Instructions for Android Studio below | Unity C++
Download google-services.json
Switch to the Project view in Android Studio to see your project root directory.
Move your downloaded google-services.json file into your module (app-level) root directory.
google-services.json
Project
MyApplication [My Application]
.gradle
.idea
app
libs
src
.gitignore
build.gradle.kts
google-services.json
proguard-rules.pro
gradle
Next
3 Add Firebase SDK

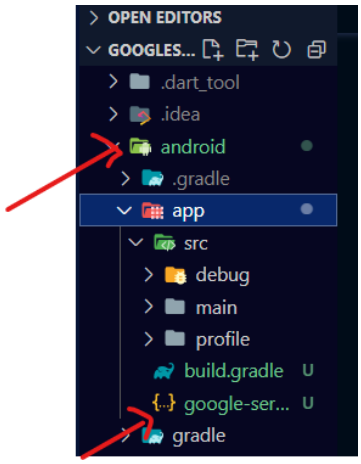

3 Add Firebase SDK
Instructions for Gradle | Unity C++
Are you still using the buildscript syntax to manage plugins? Learn how to add Firebase plugins using that syntax.
1. To make the google-services.json config values accessible to Firebase SDKs, you need the Google services Gradle plugin.
Kotlin DSL (build.gradle.kts)
Groovy (build.gradle)
Add the plugin as a dependency to your project-level build.gradle.kts file:
Root-level (project-level) Gradle file (<project>/build.gradle.kts):
plugins {
// ...
// Add the dependency for the Google services Gradle plugin
id("com.google.gms.google-services") version "4.4.1" apply false
}
2. Then, in your module (app-level) build.gradle.kts file, add both the google-services plugin and any Firebase SDKs that you want to use in your app:
Module (app-level) Gradle file (<project>/<app-module>/build.gradle.kts):


2. Then, in your module (app-level) build.gradle.kts file, add both the google-services plugin and any Firebase SDKs that you want to use in your app:
Module (app-level) Gradle file (<project>/<app-module>/build.gradle.kts):
plugins {
id("com.android.application")
// Add the Google services Gradle plugin
id("com.google.gms.google-services")
...
}
dependencies {
// Import the Firebase BoM
implementation(platform("com.google.firebase:firebase-bom:32.7.3"))
// TODO: Add the dependencies for Firebase products you want to use
// When using the BoM, don't specify versions in Firebase dependencies
implementation("com.google.firebase:firebase-analytics")
// Add the dependencies for any other desired Firebase products
// https://firebase.google.com/docs/android/setup#available-libraries
}
By using the Firebase Android BoM, your app will always use compatible Firebase library versions. Learn more
3. After adding the plugin and the desired SDKs, sync your Android project with Gradle files.
Previous
Next


Add Firebase to your Android app
Register app
Android package name: com.experiment.foodapp, App nickname: My Android App
Download and then add config file
Add Firebase SDK
4 Next steps
You're all set!
Make sure to check out the documentation to learn how to get started with each Firebase product that you want to use in your app.
You can also explore sample Firebase apps.
Or, continue to the console to explore Firebase.
Previous
Continue to console

Step 4: Let's go to your flutter project, and click on the "Android" folder, and in the App-level build gradle file you find Application id just copy it and paste it to "Android package name".

Step 5: In "App nick-name" you give any name or leave it blank because it is optional. In "Debug signing certificate SHA-1 (optional) " also leave it blank for the time or you can give the debug SHA keys. Now click on "Register app". Now you have to "Download Config file", Switch to the Project view in Android Studio/vs code to see your project root directory. Move the "google-services.json" file you just downloaded into your Android app module root directory.



Now click on "Next".

Step 6: Add Firebase SDK. The Google services plugin for Gradle loads the google-services.json file you just downloaded. Modify your build.gradle files to use the plugin. Project-level build.gradle (<project>/build.gradle):

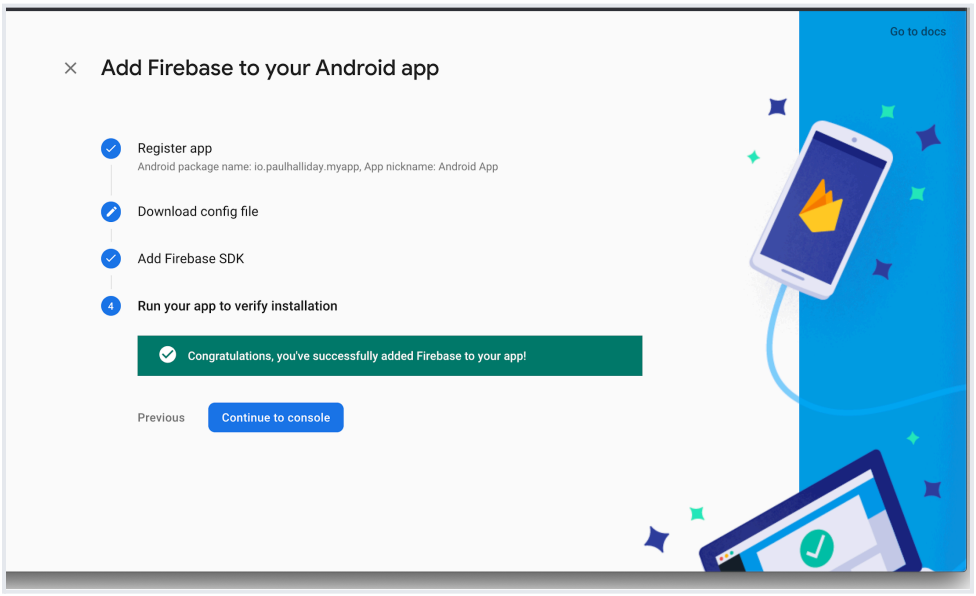
```
dependencies {
    classpath 'com.android.tools.build:gradle:4.1.0'
    classpath "org.jetbrains.kotlin:kotlin-gradle-plugin:$kotlin_version"
    classpath 'com.google.gms:google-services:4.3.8'
}
```

Step 7: App-level build.gradle (<project>/<app-module>/build.gradle):

```
apply plugin: 'com.android.application'
apply plugin: 'com.google.gms.google-services'
apply plugin: 'com.android.application'
apply plugin: 'kotlin-android'
apply from: "$flutterRoot/packages/flutter_tools/gradle/flutter.gradle"
```

```
dependencies {
    implementation "org.jetbrains.kotlin:kotlin-stdlib-jdk7:$kotlin_version"
    implementation platform('com.google.firebase:firebase-bom:28.0.1')
}
```

Step 8: After back to the firebase console and click “Next”. And then Click on “Continue to Console”.



To demonstrate with a practical example, we’ll walk you through the process of building an email-password registration and login process.

Create a Flutter and Firebase project

Create a new Flutter project using the following command:

```
flutter create flutter_authentication
```

Open the project in your favorite code editor. Here’s how to open it using VS Code:

```
code flutter_authentication
```

To integrate Firebase with your Flutter project, you have to create a new Firebase project by going to the console.

Register a new user

When a new user arrives, before logging in, they have to register to the Firebase authentication.

Create a new dart file called fire_auth.dart and define a new method called registerUsingEmailPassword():

```
class FireAuth {
 static Future<User?> registerUsingEmailPassword({
```

```
  required String name,
  required String email,
  required String password,
}) async {
  FirebaseAuth auth = FirebaseAuth.instance;
  User? user;
  try {
    UserCredential userCredential = await auth.createUserWithEmailAndPassword(
      email: email,
      password: password,
    );
    user = userCredential.user;
    await user!.updateProfile(displayName: name);
    await user.reload();
    user = auth.currentUser;
  } on FirebaseAuthException catch (e) {
    if (e.code == 'weak-password') {
      print('The password provided is too weak.');
    } else if (e.code == 'email-already-in-use') {
      print('The account already exists for that email.');
    }
  } catch (e) {
    print(e);
  }
  return user;
}
}
```

Here we are registering a new user using the email and password provided and associating the name of the user with this profile.

There can be various FirebaseAuthException errors, which we have handled in the above code snippet.

User sign-in and sign-out

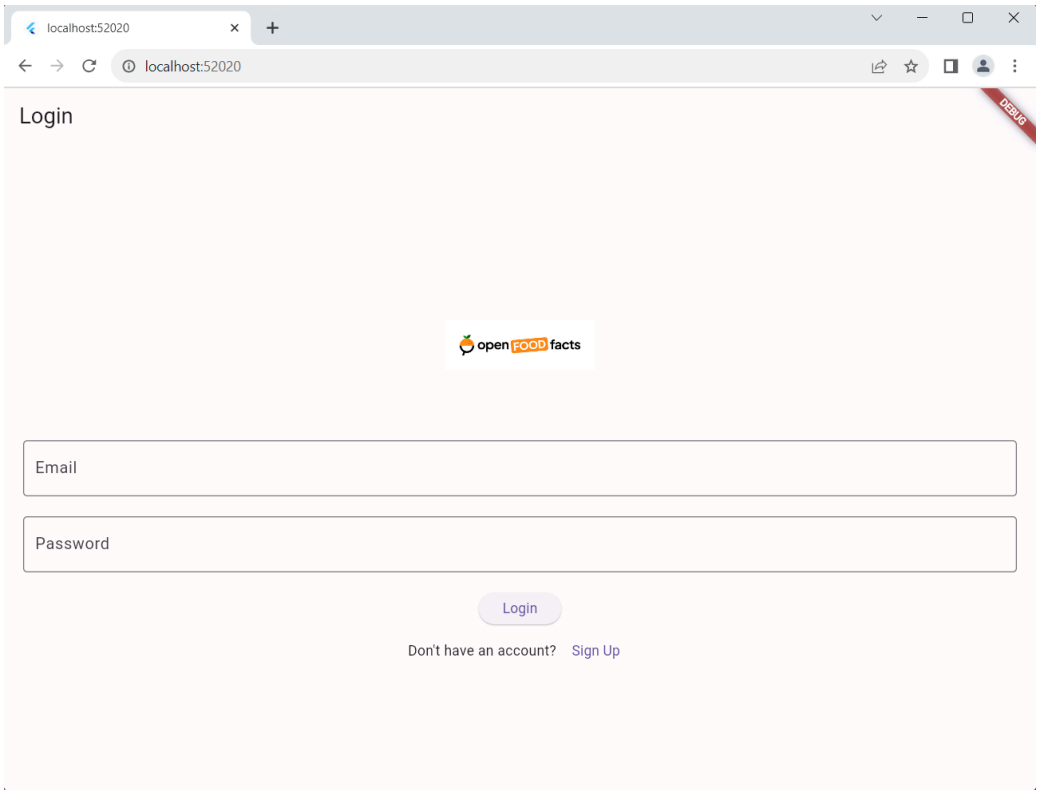
To sign in a user who has already registered in our app, define a new method called signInUsingEmailPassword(), passing the user email and password:

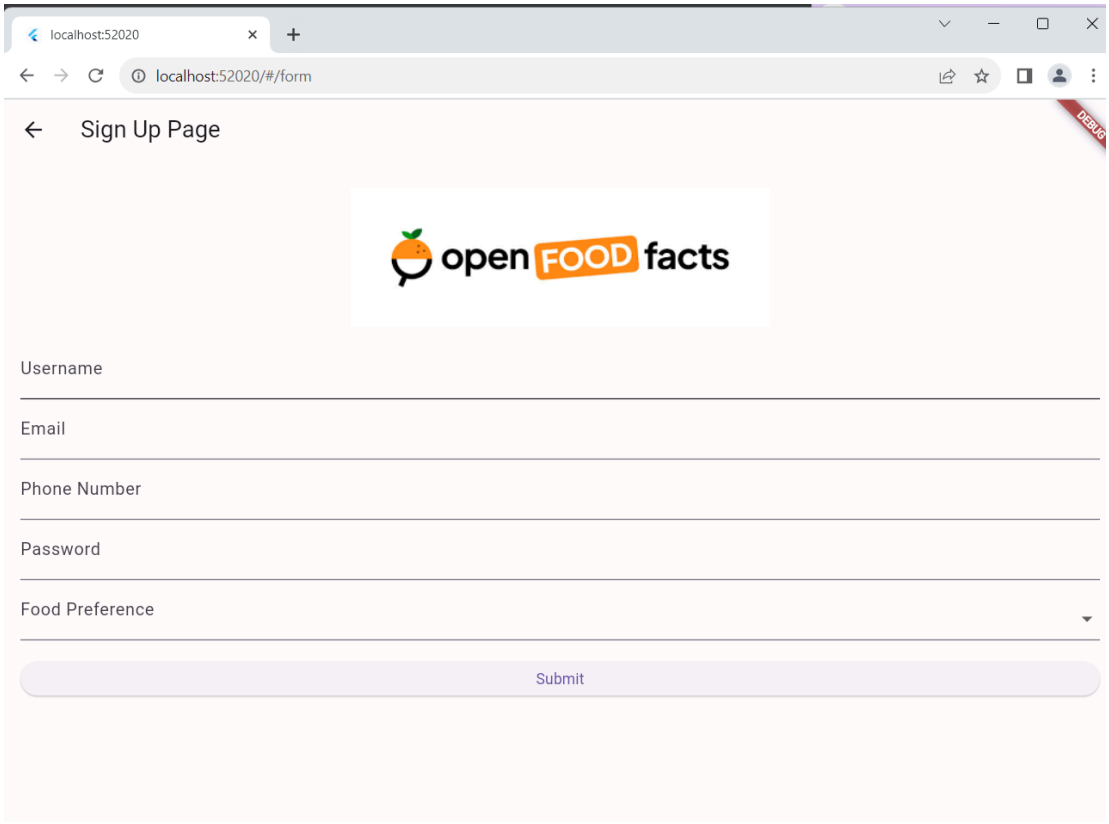
```
static Future<User?> signInUsingEmailPassword({
required String email,
required String password,
required BuildContext context,
}) async {
FirebaseAuth auth = FirebaseAuth.instance;
User? user;

try {
```

```
UserCredential userCredential = await auth.signInWithEmailAndPassword(
email: email,
password: password,
);
user = userCredential.user;
} on FirebaseAuthException catch (e) {
if (e.code == 'user-not-found') {
print('No user found for that email.');
} else if (e.code == 'wrong-password') {
print('Wrong password provided.');
}
}

return user;
}
```

The email and password are used to generate the User object provided by Firebase. The User can be used later to retrieve any additional data (e.g., user name, profile picture, etc.) stored in that account.

You can use the signOut() method to log a user out. There is no need to create a separate method for signing out because it's just a single line of code:
FirebaseAuth.instance.signOut();





Conclusion

We have successfully integrated Firebase Authentication with your Flutter app. As we may have noticed, Firebase Authentication not only provides the backend infrastructure for authenticating users easily, but also the predefined methods for auto login and email verification. And there's a lot more to explore; Firebase Authentication also provides support for integration with a number of identity providers, including Google, Facebook, Twitter, Apple, etc.

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 :
By now you must have heard this buzz word called “ Progressive Web App ”. Let me give you a simple definition about PWAs. Progressive Web Apps are Web Apps which combines the best features of Web and Native Apps. It is progressive because it is constantly progressing.

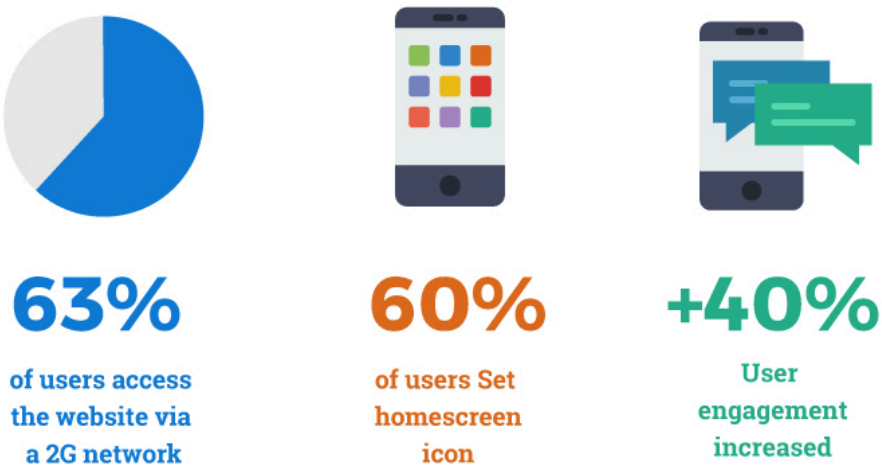
Why Progressive Web Apps?
Now let’s talk about the “WHY”.

Frances Berriman and Alex Russell came up with a concept called Progressive Web Apps? Before that let’s understand what problems does it solve.

Problems with Native Apps?
We all have used Android or iOS apps on our smartphones. We use them for all kinds of thing. But while installing any Android/iOS apps we go through these problems :

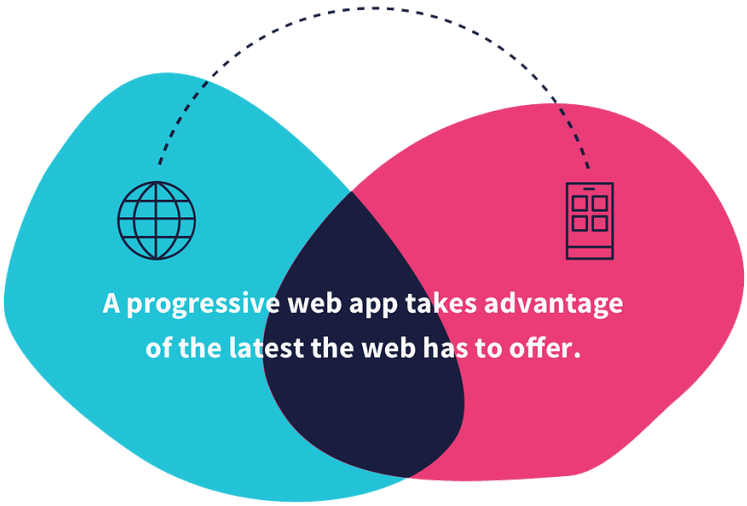
1. Is this app worth downloading?
2. Do I have enough space?
3. My available data is not sufficient.

One recent survey shows that people are turning away from Android/iOS apps, because not all app experiences are satisfying or worthwhile. Some people simply don’t want any more apps on their phone, some even hesitate to download any app.
If you take a look at the apps installed on your mobile right now there might be at least a dozen apps that you do not use regularly. Sometimes apps only works good when the phone has an active internet connection.



The irony is that most of the apps have a fully responsive website performing the same functions. So why waste your precious disk space and your internet data on your smartphone by

installing the native app? The average size of apps that we install from play store/app stores would range from 30–200MB. Moreover, these app needs to updated every week! But Progressive Web Apps are within some KBs and are automatically updated. Thanks to service worker.
What if a website can do that and much more than a Native app? This is what Progressive Web Apps (PWA) are trying to accomplish.



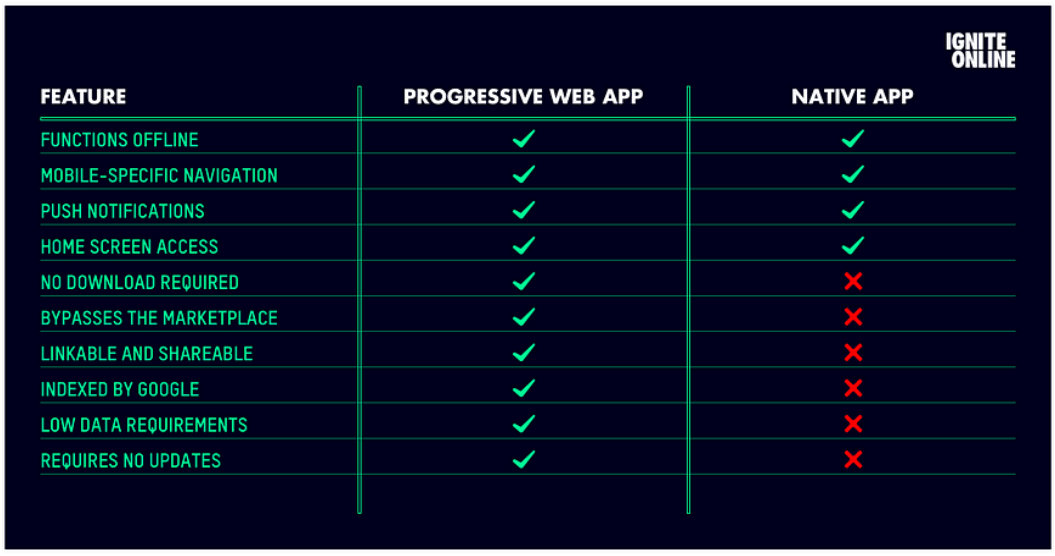
In short, Progressive web apps combine everything that is great about a native mobile application with everything that is great about a mobile website.
Some other ways I like to describe them:
“The best of the web, plus the best of native apps”
Or, in Alex’s words:
“Just websites that took all the right vitamins”

Features of PWAs:

- Progressive — The word progressive means it works for every user, regardless of browser choice because they're built with progressive enhancement as a core tenet.
- Responsive — Automatically adjustable to any form: desktop, mobile, tablet etc.
- Load Time — Progressive Web Apps are instantly available
- App-like — Feels like a mobile app with app-style interactions since it's built on the app shell model.
- Fresh — Always up-to-date so you do not need to update it again and again like any other Android/iOS apps.
- Safe — Served via HTTPS to ensure content is securely delivered
- Engaging — Features like push notifications, etc. makes it very engaging.
- Installable — Allows users to install the website as an app on their home screen without taking the user to an app store.
- Linkable — Easily shared via a URL and does not require complex installation.

Benefits of making a Progressive Web App rather than building a fully functional Android App?

- Cost Effective — For an app publisher, the biggest advantage is the cost saving in terms of app development and maintenance. Because it is assumed that making a website is a lot easier than making an Android App.
- Cross Platform — Unlike any other apps, Progressive Web Apps are not restricted to any specific platform. That means you do not need to develop separate versions of app for different platforms.

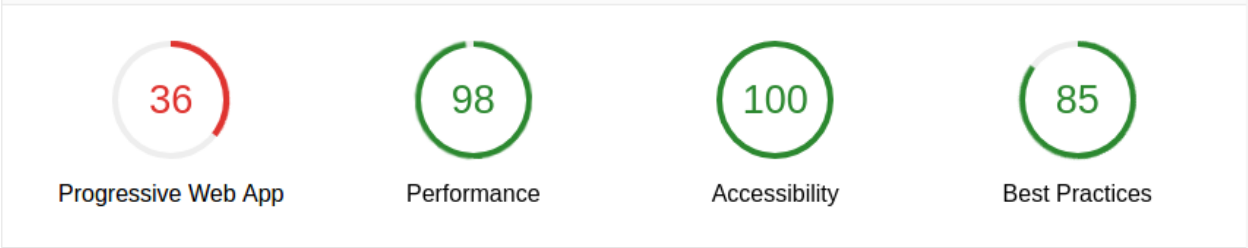


Some Popular Companies that Do Progressive Web Apps

- Ola
- Flipkart
- pinterest
- Twitter

- Alibaba
- BookMyShow
- MakeMyTrip
- OLX
- The Weather Channel
- Forbes
- JioCinema
- Trivago

Now Let's create a Progressive Web App:
Here, I'll show you how I've created a Progressive Web App for simple blog. You can get the codes here. Now that we have a basic website we can start turning it into a progressive web app. To do this we need to add a few things to it which I'll go through as we need them.
Testing your PWA
To check if your site is working as a PWA you can use Lighthouse. Lighthouse is a chrome extension that will tell you if your site is a good PWA and if it isn't how to improve it.
Once installed open up your website and click on the Lighthouse icon in the top right of your browser then Generate Report.



Lighthouse results before I started working on the PWA parts of the site
Make an app icon
Your site is going to be on a home screen, you need some sort of icon to represent it. Here I've used a downloaded logo from internet.

Register Service Worker
Add service worker <script> to index.html:

A service worker is another file we add to our project, it will allow the site to work offline. It is also a requirement of a PWA, so we need one.

service-worker.js

```
self.addEventListener("install", function (event) {
    event.waitUntil(preLoad());
});

self.addEventListener("fetch", function (event) {
    event.respondWith(checkResponse(event.request).catch(function () {
        console.log("Fetch from cache successful!")
        return returnFromCache(event.request);
    }));
    console.log("Fetch successful!")
    event.waitUntil(addToCache(event.request));
});
self.addEventListener('sync', event => {
    if (event.tag === 'Sync from cache') {
        console.log("Sync successful!")
    }
});
self.addEventListener('push', function (event) {
    if (event && event.data) {
        var data = event.data.json();
        if (data.method == "PushMessageData") {
            console.log("Push notification sent");
            event.waitUntil(self.registration.showNotification("RED
STORE", {body: data.message}))
        }
    }
})

var filesToCache = [
    '/index.html',
    '/product_details.html',
    '/products.html',
    '/cart.html',
    '/account.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("index.html");
            } else {
                return matching;
            }
        });
    });
};
```

Web App Manifest

To support add to homescreen feature, we need to create a manifest file.

“ The web app manifest provides information about an application (such as name, author, icon, and description) in a JSON text file. The purpose of the manifest is to install web applications to the homescreen of a device, providing users with quicker access and a richer experience.“

The simplest way I did this was by using this online icon generator tool. Feed it your shiny new icon and it will spit out a bunch of resized versions and some HTML code.

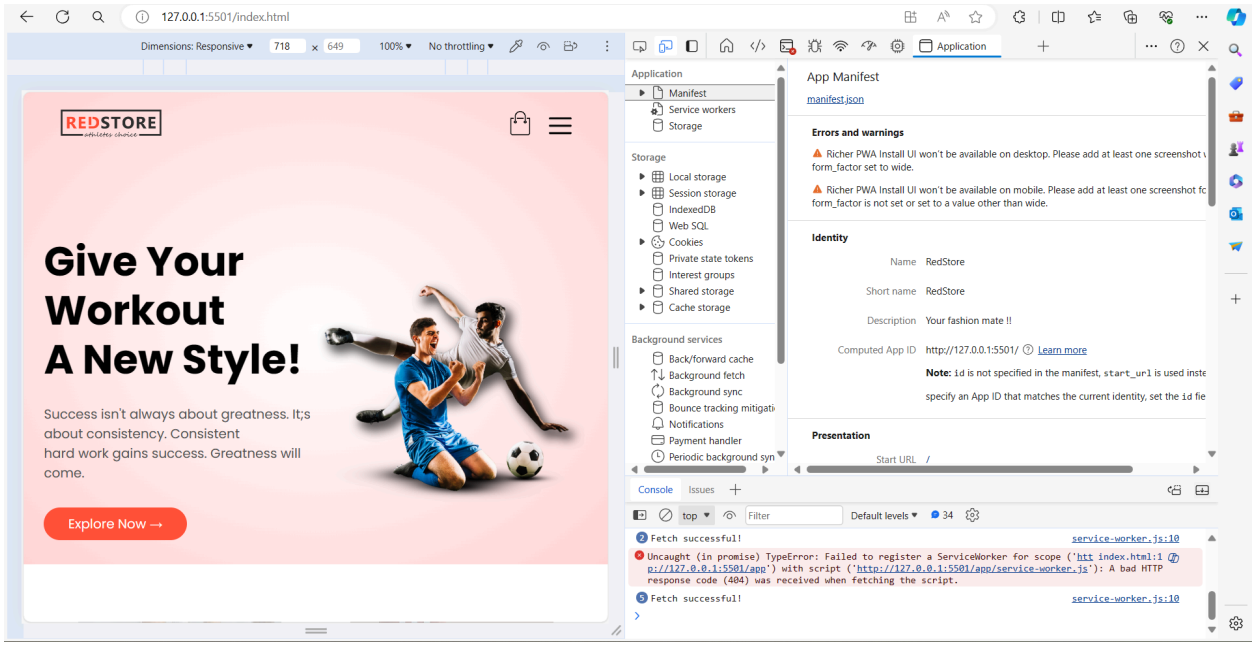
- Download the file it gives you and unzip it.
- Put the icons in a folder next to the rest of your site.
- Add the code it gave you to the <head> of you index.html file
- Make sure the path to the icons is right. I put them all in a sub folder so had to add “icons/” to each line.

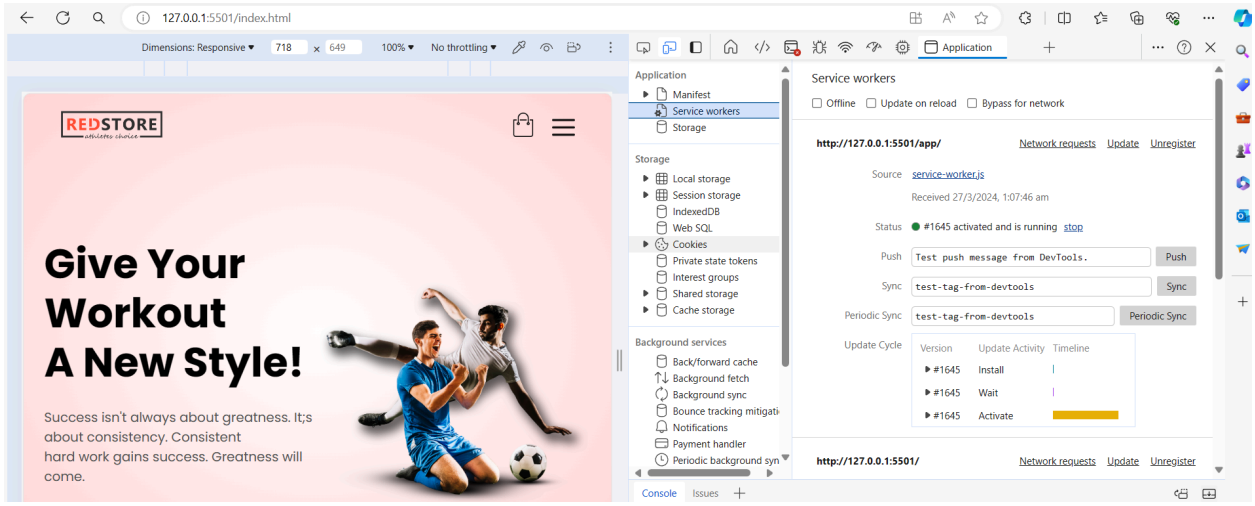
manifest.json

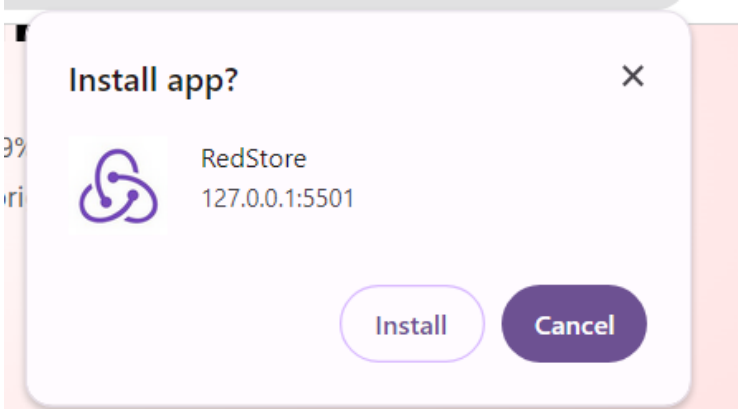
```
{
    "name": "RedStore",
    "short_name": "RedStore",
    "start_url": "/",
    "background_color": "#212529",
    "description": "Your fashion mate !!",
    "display": "fullscreen",
    "theme_color": "#212529",
    "icons": [
        {
            "src": "/icon2.png",
            "sizes": "512x512",
            "purpose": "any"
        },
        {
            "src": "/icon1.png",
            "sizes": "192x192",
```

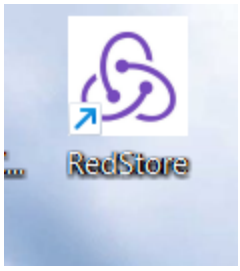
```
            "purpose": "any"
        },
        {
            "src": "/icon1.png",
            "sizes": "192x192",
            "purpose": "maskable"
        },
        {
            "src": "/icon1.png",
            "sizes": "192x192",
            "purpose": "maskable"
        }

    ],
    "prefer_related_applications": true,
    "related_applications": [
        {
            "platform": "play",
            "id": "com.google.samples.apps.iosched"
        }
    ]
}
```

Now you have a manifest that was created by the icon generator tool, but we're going to add a little bit more to it.
Head over to a web app manifest generator and start filling in the info about your site. After doing all these things my manifest ended up looking like this.


127.0.0.1:5501/index.html
REDSTORE
Give Your Workout A New Style!
Success isn't always about greatness. It;s about consistency. Consistent hard work gains success. Greatness will come.
Explore Now →
Application
Manifest
Service workers
Storage
App Manifest
manifest.json
Errors and warnings
Identity
Name RedStore
Short name RedStore
Description Your fashion mate !!
Presentation
Console
Issues
Fetch successful!
service-worker.js:10


127.0.0.1:5501/index.html
REDSTORE
Give Your Workout A New Style!
Success isn't always about greatness. It;s about consistency. Consistent hard work gains success. Greatness will come.
Application
Service workers
Offline
Update on reload
Bypass for network
http://127.0.0.1:5501/app/
Network requests
Update
Unregister
Source service-worker.js
Status #1645 activated and is running stop
Push
Sync
Periodic Sync
Update Cycle
Version
Update Activity
Timeline
Install
Wait
Activate
http://127.0.0.1:5501/


Install app?
RedStore
127.0.0.1:5501
Install
Cancel



Conclusion : By following these steps and providing the necessary metadata in our Web App Manifest file, you can successfully enable the "Add to Home Screen" feature for our eCommerce PWA, improving user accessibility and engagement.

EXPERIMENT : 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

A service worker is a web worker. It is a JavaScript file that runs aside from the mobile browser. In other words, it is another technical component that promotes the functionality of PWA. The service worker retrieves the resources from the cache memory and delivers the messages.
It is independent of the application to which they are connected, and has many consequences:

- The service worker does not block the synchronized XHR, so it cannot be used in local storage (It is designed to be completely asynchronous).
- It can receive information from the server even when the application is not running. It shows notifications in the PWA application without opening in the mobile browser.
- It cannot directly access the DOM. Therefore, the PostMessage and Message Event Listener method is used to communicate with the webpage. The PostMessage method is used to send data, and the message event listener is used to receive data.

Things to understand about it:

- It is a programmable-network proxy that helps you monitor how your page handles network requests.
- It only works on the HTTPS because HTTPS is very secure, and it intercepts network requests and modifies responses.

Service worker lifecycle

The service worker lifecycle is the most complex part of the PWA. There are three stages in the lifetime of a service worker:

1. Registration
2. Installation
3. Activation

Registration

A service worker is basically a JavaScript file. You need to register it in your crucial JavaScript code to use a service worker. Registration tells your browser location of the service worker and starts installing it on the background. One thing that distinguishes a service worker file from a standard JavaScript file is that a service worker runs in the background away from the mobile's main browser. This process is the first phase of the service worker's lifecycle.
The code of service worker registration is placed in the main.js file.

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

First, this code checks whether the browser supports the service worker. The service worker is then registered with navigator.serviceWorker.register when the service worker returns a promise. If the promise is fulfilled, the registration is successful; otherwise, the registration is failed.
Scope of registration
This scope determines the web pages that are managed by a service worker. The location of the service worker defines the default scope. Whenever you register a service worker file at the main folder of the system, it is not important to specify its scope, because it controls all webpages.

1. navigator.service.worker.register ('/sw.js', { scope: '/' });

Installation
When a new service worker is registered with the help of navigator.serviceWorker.register, the JavaScript code is downloaded, and the installation state is entered. If the installation succeeds, the service worker further proceeds to the next state.

```
const assets to.cache = [
'/js/app.js',
'/about.html',
'/index.html',
'/css/app.css',
]

self. addEventListener ('install', function (event) {
   event. wait Until (
      caches. Open('staticAssetsCache')
      then (function (cache) {
          return cache Add All (assetsToCache.);
      })
```

```
    );
});
```

Activation

Once the service worker is successfully installed, it converts to the activation phase. If there is an open page controlled by the previous service worker, the new service worker enters the waiting state. The new service worker is activated only when no pages are loaded in the old service worker. A service worker is not activated immediately after installation.

A service worker will only be active in these cases:

- If no service worker currently active.
- If the user refreshes the webpage.

Service-worker.js

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

The service worker can manage network requests rather than caching. It roves around the latest Internet API.

1. Fetch: The Fetch API is a basic resource of the GUI. It makes it easier to control webpage requests and responses than older XMLHttprequests, and this often needs extra syntax, and its example is controlling the redirects. When a resource is requested on the network than the fetal event is terminated.

Note: It supports the CORS (Cross-Origin Resource Sharing). A local server is usually required for testing.

Fetch request example:

```
fetch('ABCs/ABC.json')
then (function (response){
// response function
})
catch (function (error)
{
console.log ('problem section: \n', error);
});
```

2. Cache API: A cache interface has been provided for the service worker API, which allows you to create a repository of responses as requested. However, this interface was designed for service workers. It does not update the memory in the cache unless specifically requested.

Features of the service worker

- **Offline:** Enabling offline functionality is possibly the most demanding service worker facility.
- **Caching:** The control of cache content is the most common feature for service workers.
- Content delivery networks: The CDN and other external material may be difficult to handle. The PWA developers sometimes select out of publicly hosted software due to same-origin rules and SSL, but you may still upload scripts from CDNs.
- **Push notifications:** The feature of push notifications is the best way to interact with users and visitors. This feature enhances the performance of the PWA application.
- **Background synchronization:** It is another very important feature of a service worker that synchronizes tasks in the background.

4. Webpack

The webpack is the fourth component of the PWA. It is used to design the PWA front-end. It allows the PWA-developers to gather all JavaScript resources and data in one location.

5. Transport Layer Security (TLS)

The transport layer security is the fifth component of the PWA. This component is a standard for all robust and secure data exchange between any two applications. The integrity of the data requires the website's service through the HTTPS and an SSL certificate installed on the server.

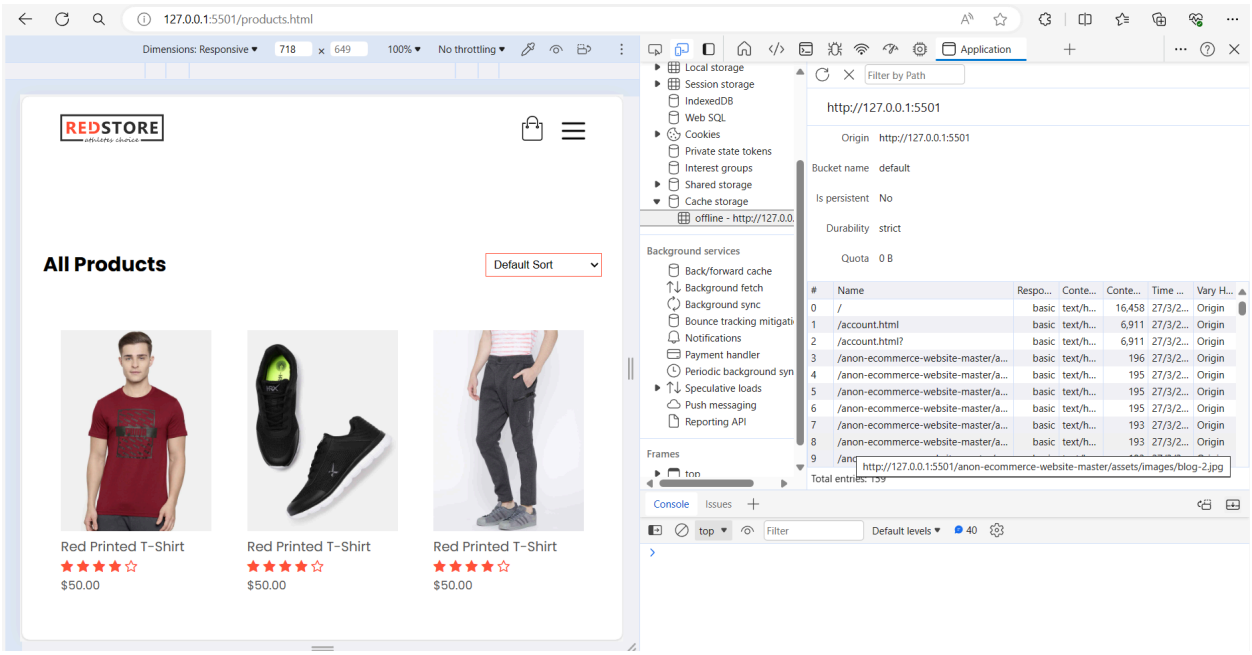
service-worker.js

```
self.addEventListener("install", function (event) {
    event.waitUntil(preLoad());
});

self.addEventListener('sync', event => {
    if (event.tag === 'Sync from cache') {
        console.log("Sync successful!")
    }
});

self.addEventListener('push', function (event) {
    if (event && event.data) {
        var data = event.data.json();
        if (data.method == "PushMessageData") {
            console.log("Push notification sent");
            event.waitUntil(self.registration.showNotification("RED
STORE", { body: data.message }))
        }
    }
});
```

```
var filesToCache = [
    '/index.html',
    '/product_details.html',
    '/products.html',
    '/cart.html',
    '/account.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("index.html");
            } else {
                return matching;
            }
        });
    });
};

// Add event listener for beforeinstallprompt
self.addEventListener('beforeinstallprompt', event => {
    event.preventDefault();
    const installButton = document.getElementById('install-button');
    if (installButton) {
        installButton.style.display = 'block';
        installButton.addEventListener('click', () => {
            event.prompt();
        });
    }
});
```

OUTPUT:



Conclusion: By following these steps and best practices, we have effectively code, register, and manage a service worker for the E-commerce PWA, providing users with a seamless and reliable shopping experience both online and offline.

EXPERIMENT : 9

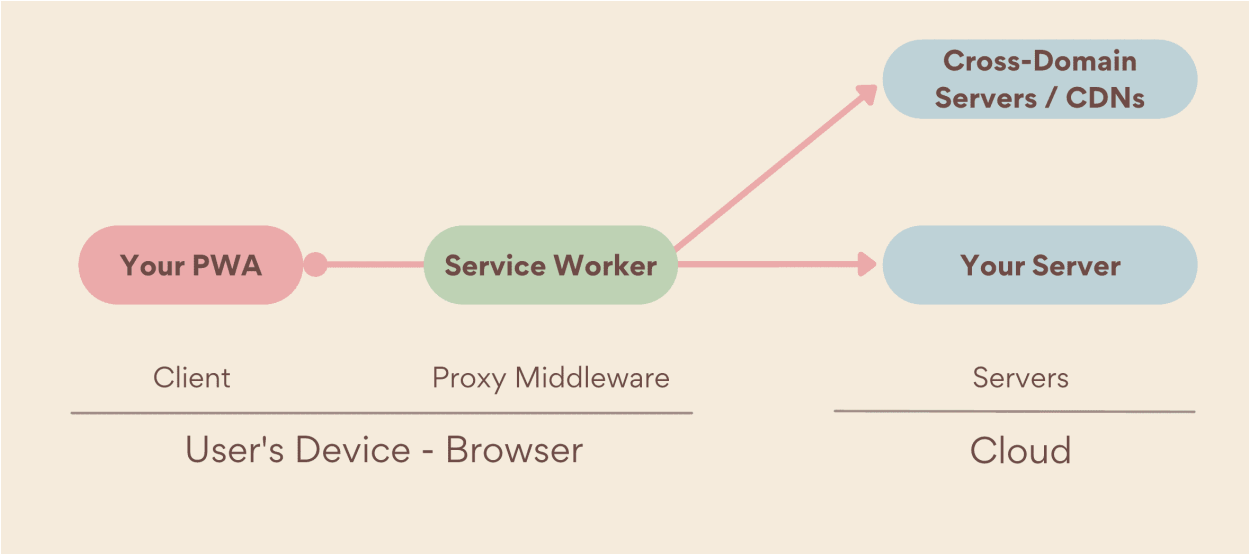
Aim : To implement Service worker events like fetch, sync and push for E-commerce PWA

Theory :

Service workers

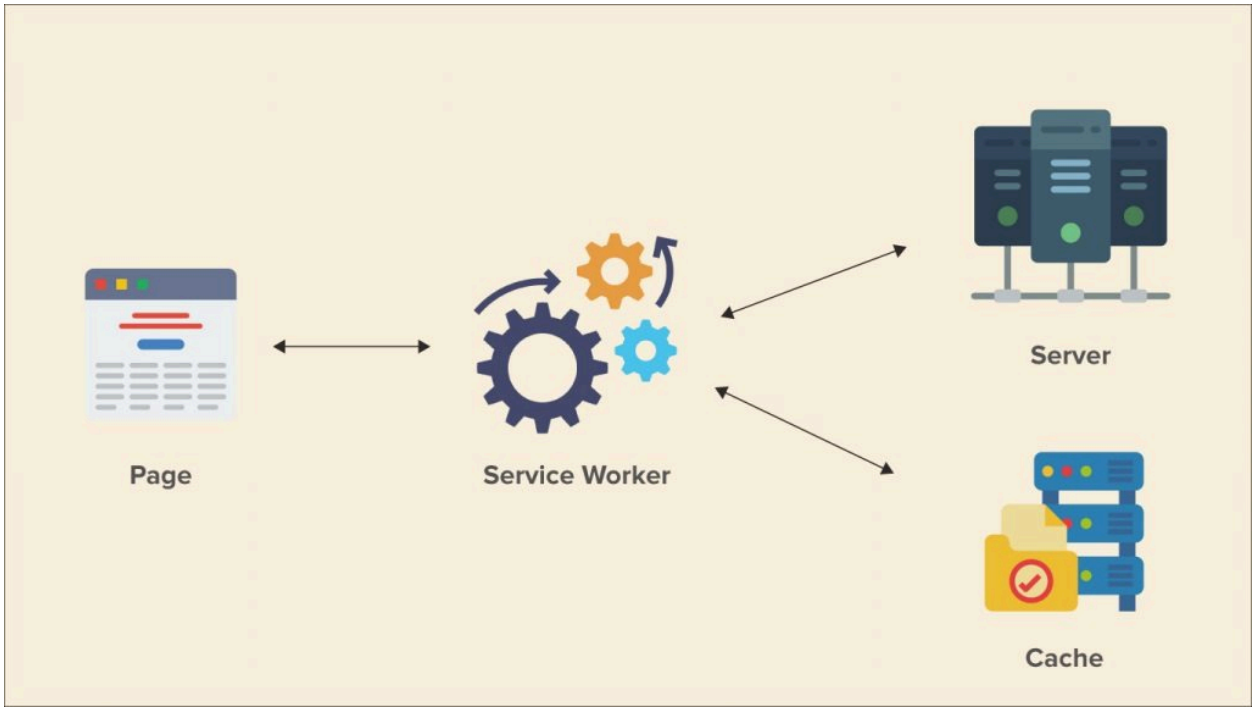
Users expect apps to start on slow or flaky network connections, or even when offline. They expect the content they've most recently interacted with, such as media tracks or tickets and itineraries, to be available and usable. When a request isn't possible, they expect the app to tell them instead of silently failing or crashing. And users wish to do it all quickly. As we can see in this study Milliseconds make millions, even a 0.1 second improvement in load times can improve conversion by up to 10%. In summary: users expect PWAs to be reliable and that's why we have service workers.

Hello service workers



When an app requests a resource covered by the service worker's scope, including when a user is offline, the service worker intercepts the request, acting as a network proxy. It can then decide if it should serve the resource from the cache via the Cache Storage API, from the network as normally would happen without a service worker, or create it from a local algorithm. This lets you provide a similar experience to that provided by a platform app. It can even work entirely off line.

Not all browsers support service workers. Even when present your service worker won't be available on first load or while it's waiting to activate. Therefore, treat it as optional and do not require it for core functionality.



Registering a service worker

Before a service worker takes control of your page, it must be registered for your PWA. That means the first time a user comes to your PWA, network requests will go directly to your server because the service worker is not yet in control of your pages.
After checking if the browser supports the Service Worker API, your PWA can register a service worker. When loaded, the service worker sets up shop between your PWA and the network, intercepting requests and serving the corresponding responses.

```
if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register("/serviceworker.js");
}
```

Note: There is only one service worker per PWA, but that doesn't mean you need to place the code only in one file. A service worker can include other files using importScripts in every browser or using ECMAScript module imports in some modern browsers.

Verify if a service worker is registered

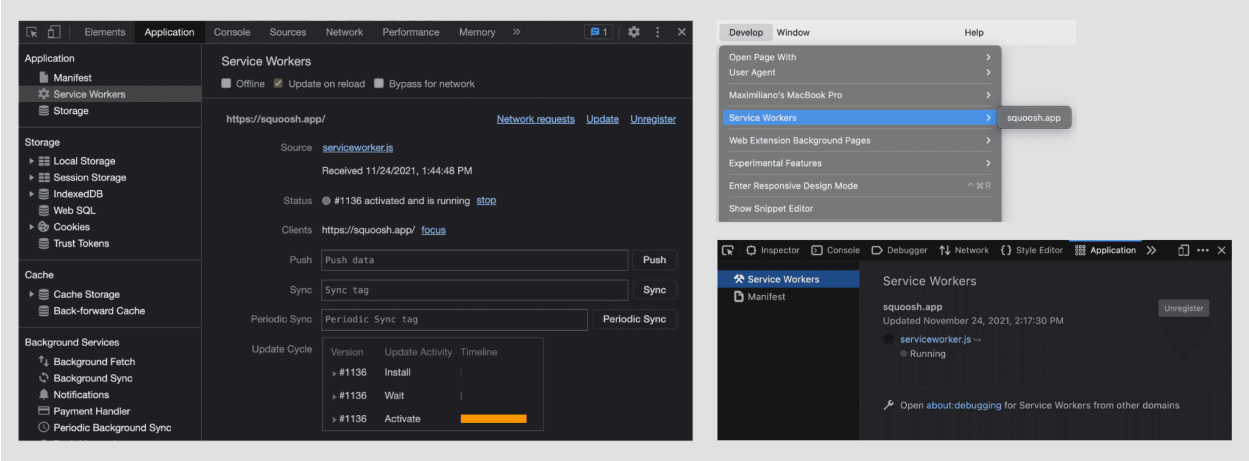
To verify if a service worker is registered, use developer tools in your favorite browser.

In Firefox and Chromium-based browsers (Microsoft Edge, Google Chrome, or Samsung Internet):

1. Open developer tools, then click the Application tab.
2. In the left pane, select Service Workers.
3. Check that the service worker's script URL appears with the status "Activated". (You'll learn what this status means in the lifecycle section in this chapter). On Firefox the status can be "Running" or "Stopped".

In Safari:

1. Click the Develop menu then the Service Workers submenu.
2. Check that an entry with the current origin appears in the submenu. It opens an inspector over the service worker's context.



Scope

The folder your service worker sits in determines its scope. A service worker that lives at example.com/my-pwa/sw.js can control any navigation at the my-pwa path or below, such as example.com/my-pwa/demos/. Service workers can only control items (pages, workers, collectively "clients") in their scope. Scope applies to browser tabs and PWA windows.

Only one service worker per scope is allowed. When active and running, only one instance is typically available no matter how many clients are in memory (such as PWA windows or browser tabs).

Warning: You should set the scope of your service worker as close to the root of your app as possible. This is the most common setup as it lets the service worker intercept all the requests related to your PWA. Don't put it inside, for instance, a JavaScript folder or have it loaded from a CDN.

Safari has more complex scope management, known as partitions, affecting how scopes work if you have cross-domain iframes. To read more about WebKit's implementation, read their blog post.

Lifecycle

Service workers have a lifecycle that dictates how they are installed, this is separate from your PWA installation. The service worker lifecycle starts with registering the service worker. The browser then attempts to download and parse the service worker file. If parsing succeeds, its install event is fired. The install event only fires once.
Service worker installation happens silently, without requiring user permission, even if the user doesn't install the PWA. The Service Worker API is even available on platforms that do not support PWA installation, such as Safari and Firefox on desktop devices.
Service worker registration and installation, while related, are different events. Registration happens when a page requests a service worker by calling register() as described previously. Installation happens when a registered service worker exists, can be parsed as JavaScript, and doesn't throw any errors during its first execution.
After the installation, the service worker is not yet in control of its clients, including your PWA. It needs to be activated first. When the service worker is ready to control its clients, the activate event will fire. This doesn't mean, though, that the page that registered the service worker will be managed. By default, the service worker will not take control until the next time you navigate to that page, either due to reloading the page or re-opening the PWA.

You can listen for events in the service worker's global scope using the self object.
serviceworker.js

```
// This code executes in its own worker or thread
self.addEventListener("install", event => {
  console.log("Service worker installed");
});
self.addEventListener("activate", event => {
  console.log("Service worker activated");
});
```

Updating a service worker

Service workers get updated when the browser detects that the service worker currently controlling the client and the new (from your server) version of the same file are byte-different.

Warning: When updating your service worker, do so without renaming it. Do not even add file hashes to the filename. Otherwise, the browser will never get the new version of your service worker!
After a successful installation, the new service worker will wait to activate until the existing (old) service worker no longer controls any clients. This state is called "waiting", and it's how the browser ensures that only one version of your service worker is running at a time. Refreshing a page or reopening the PWA won't make the new service worker take control. The user needs to close or navigate away from all tabs and windows using the current service worker and then navigate back. Only then will the new service worker take control.

Service worker lifespan

Once installed and registered, a service worker can manage all network requests within its scope. It runs on its own thread, with activation and termination controlled by the browser. This lets it work even before or after your PWA is open. While service workers run on their own thread, there is no guarantee that in-memory state will persist between runs of a service worker, so make sure anything you want to reuse on each run is available either in IndexedDB or some other persistent storage.
If not already running, a service worker will start whenever a network request in its scope is asked for, or when a triggering event, like periodic background sync or a push message, is received.
Service workers don't live indefinitely. While exact timings differ between browsers, service workers will be terminated if they've been idle for a few seconds, or if they've been busy for too long. If a service worker has been terminated and an event occurs that would start it up, it will restart.

Capabilities

With a registered and active service worker, you have a thread with a completely different execution lifecycle than the main one on your PWA. However, by default, the service worker file itself has no behavior. It won't cache or serve any resources, as this has to be done by your code. You'll find out how in the following chapters.
Service worker's capabilities are not just for proxy or serving HTTP requests; other features are available on top of it for other purposes, such as background code execution, web push notifications, and process payments.

service-worker.js

```
self.addEventListener("install", function (event) {
```

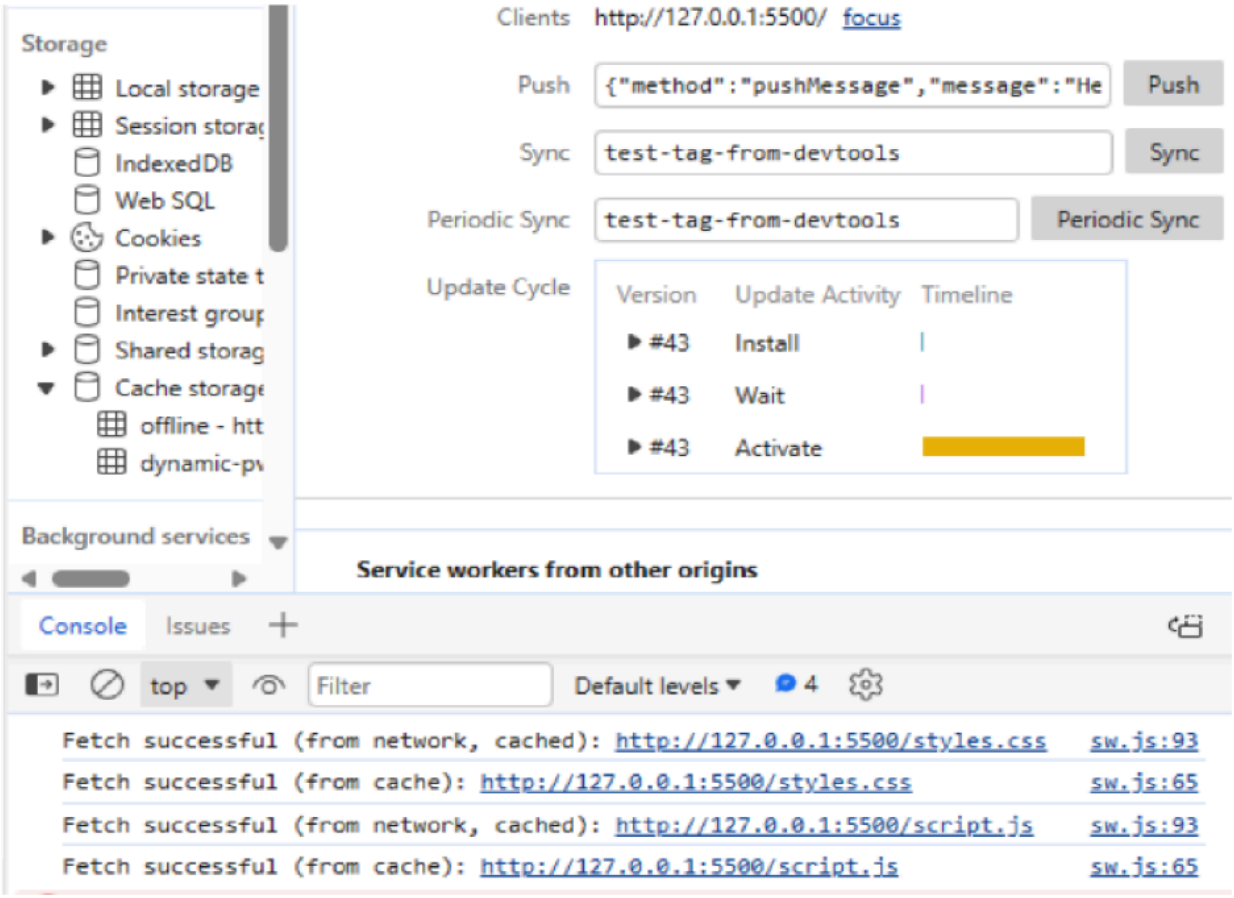
```
    event.waitUntil(preLoad());
});

self.addEventListener('sync', event => {
    if (event.tag === 'Sync from cache') {
        console.log("Sync successful!")
    }
});

self.addEventListener('push', function (event) {
    if (event && event.data) {
        var data = event.data.json();
        if (data.method == "PushMessageData") {
            console.log("Push notification sent");
            event.waitUntil(self.registration.showNotification("RED
STORE", { body: data.message }))
        }
    }
});

var filesToCache = [
    '/index.html',
    '/product_details.html',
    '/products.html',
    '/cart.html',
    '/account.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("index.html");
            } else {
                return matching;
            }
        });
    });
};

// Add event listener for beforeinstallprompt
self.addEventListener('beforeinstallprompt', event => {
    event.preventDefault();
    const installButton = document.getElementById('install-button');
    if (installButton) {
        installButton.style.display = 'block';
```

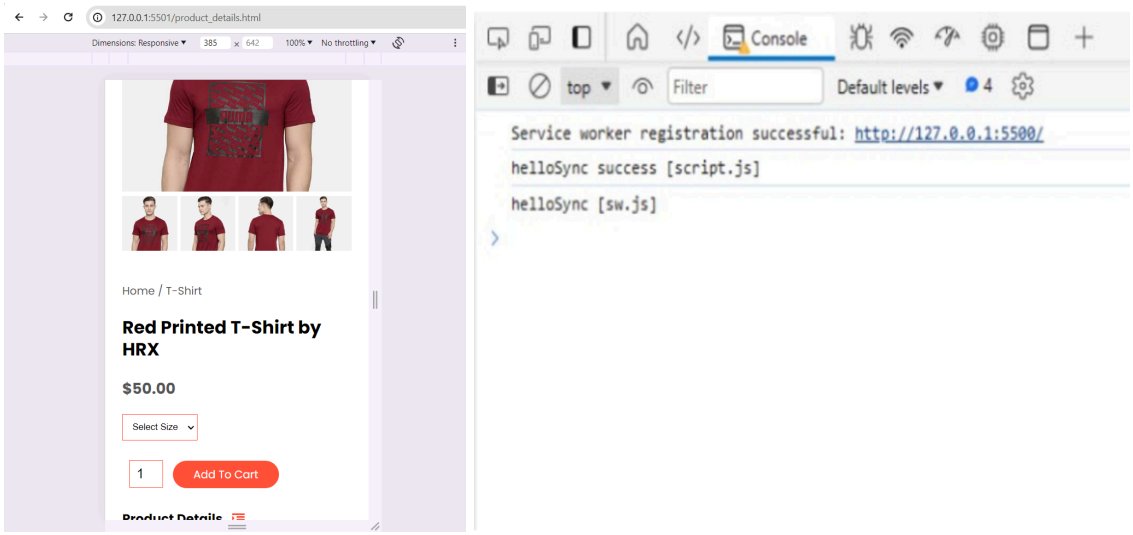
```
        installButton.addEventListener('click', () => {
            event.prompt();
        });
    }
});
```

Output:

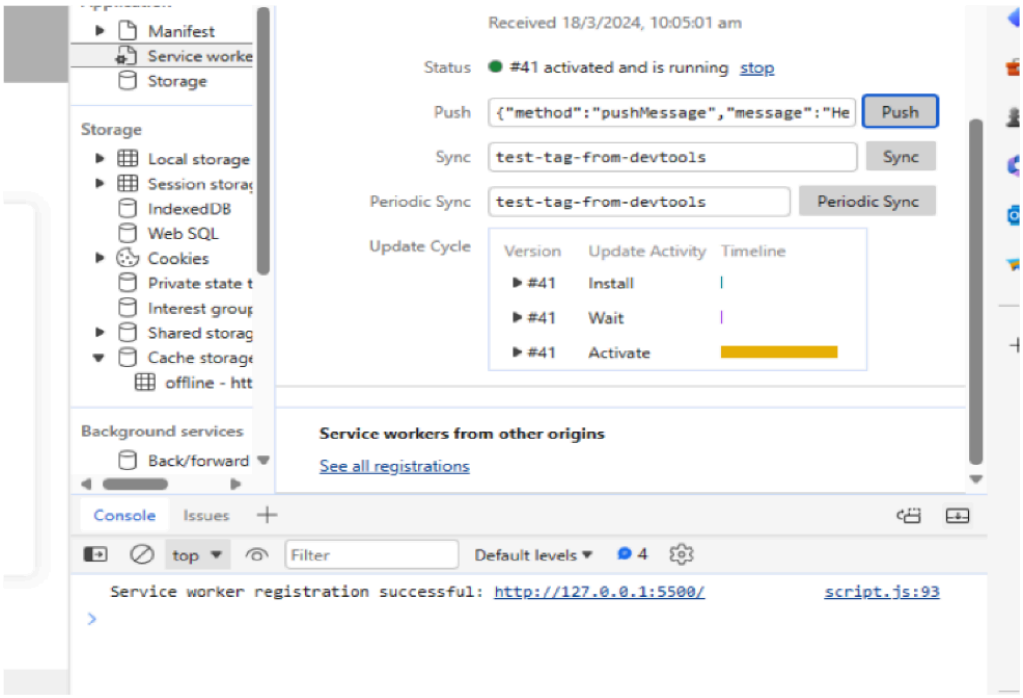
Fetch Event



Sync Event



Push Event



Conclusion : Therefore, integrating service worker events fetch, sync, and push into the E-commerce PWA enhances its functionality, performance, and user engagement, making it a powerful and reliable platform for online shopping experiences.

EXPERIMENT : 10

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

Theory :
GitHub Pages is a platform that allows users to host public web pages directly from their GitHub repositories. It offers several key features, including:

1. Blogging with Jekyll: GitHub Pages supports Jekyll, a static site generator, making it easy to create and maintain blogs.
2. Custom URL: Users can set up custom domain names for their GitHub Pages sites.
3. Automatic Page Generator: GitHub Pages provides an automatic page generator for quickly creating basic websites.

Reasons for favoring GitHub Pages over Firebase include:

1. Free to use: GitHub Pages is free to use for public repositories.
2. Integrated with GitHub: Since GitHub Pages is integrated with GitHub, it's convenient for users already familiar with the platform.
3. Quick setup: Setting up a GitHub Pages site is straightforward, requiring only the push of static website files to a specific branch.

GitHub Pages is trusted by companies like Lyft, CircleCI, and HubSpot, and is popular among developers, with a significant presence in both company and developer stacks.

Pros of GitHub Pages:
1. Familiar interface for GitHub users.
2. Easy setup process by pushing static website files.
3. Built-in support for Jekyll and custom domains.
4. Custom domain support with CNAME file configuration.

Cons of GitHub Pages:
1. Public visibility of website code unless using a private repository.
2. Limited HTTPS support for custom domains.
3. Inconsistent support for Jekyll plugins.

Firebase, on the other hand, is a real-time app platform designed to power collaborative applications. Some of its features include:

1. Real-time data synchronization across clients and the Firebase cloud.
2. Client-side development with real-time updates.
3. JSON-based data storage with URL accessibility for easy integration and viewing.

Reasons for favoring Firebase over GitHub Pages:
1. Real-time backend functionality.
2. Fast and responsive data synchronization.
3. Hosted by Google with access to additional services like authentication and cloud messaging.

Popular companies such as Instacart, 9GAG, and Twitch use Firebase, and it's mentioned in numerous company and developer stacks.

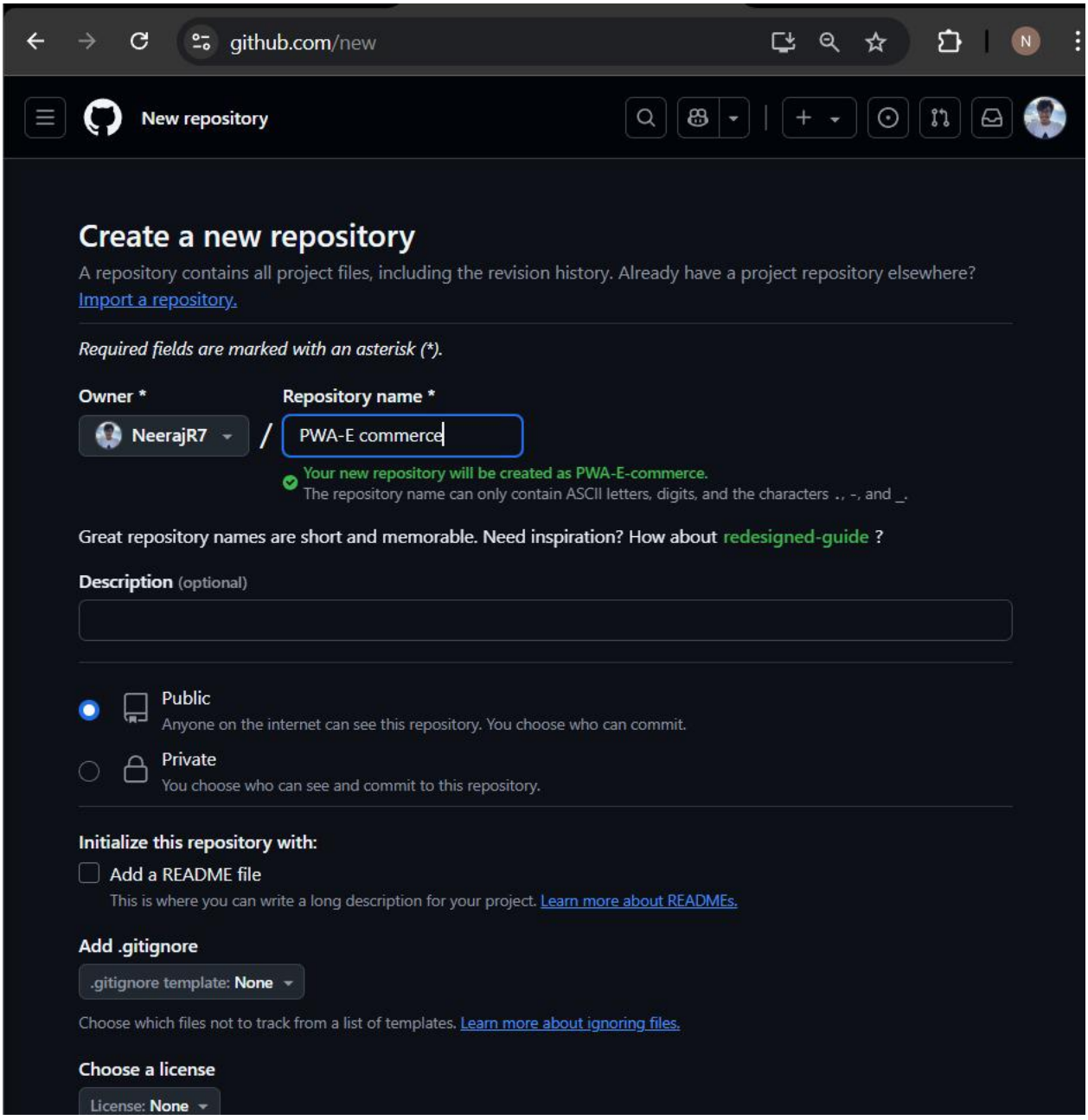
Pros of Firebase:
1. Hosted by Google with access to various services.
2. Provides authentication, cloud messaging, and real-time database functionality.
3. Support for HTTPS and provision of free SSL certificates for custom domains.

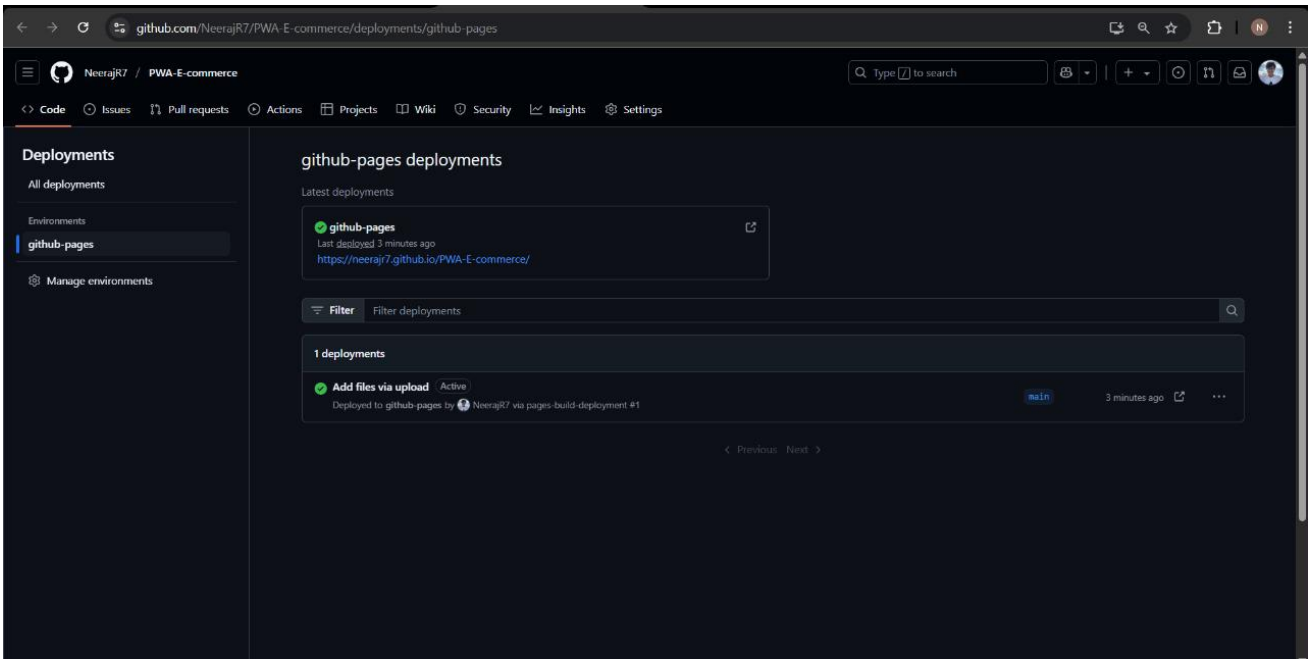
Cons of Firebase:
1. Limited data transfer allowance per month.
2. Command-line interface only for management.
3. Lack of built-in support for static site generation.

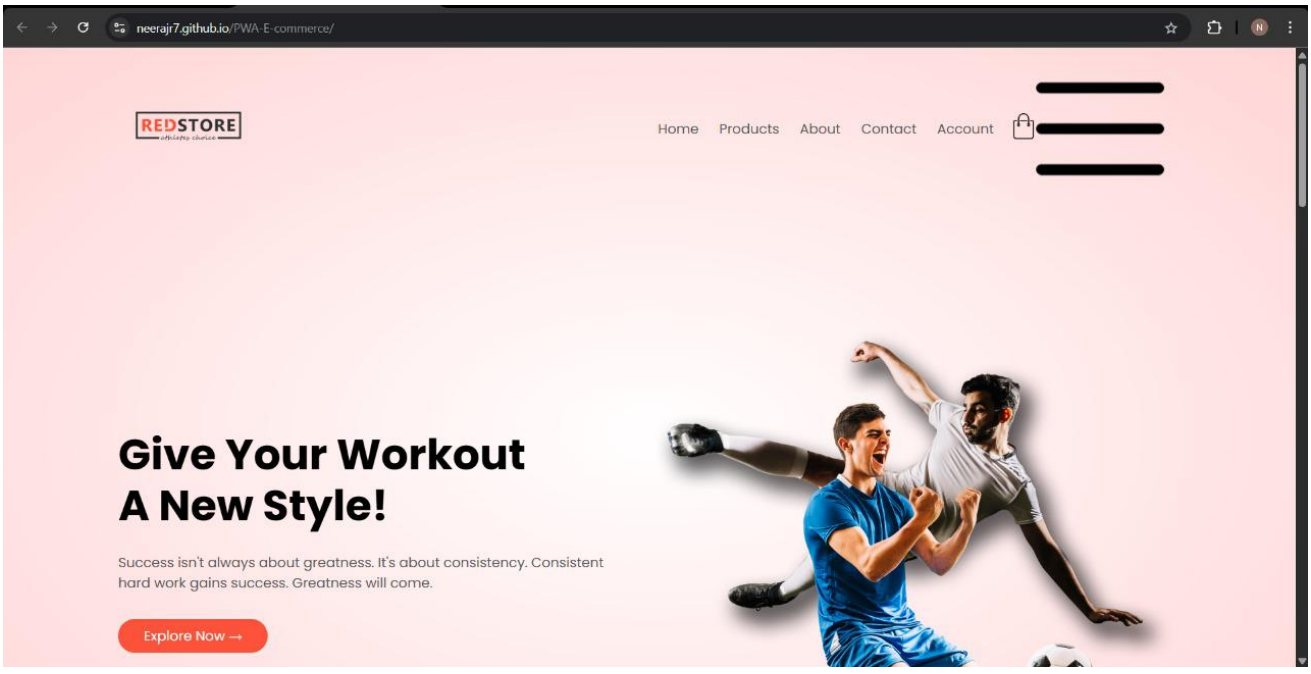
The hosted GitHub repository link for PWA project is:

https://neerajr7.github.io/PWA-E-commerce/

Output:


github.com/new
New repository
Create a new repository
A repository contains all project files, including the revision history. Already have a project repository elsewhere?
Import a repository.
Required fields are marked with an asterisk (*).
Owner *
Repository name *
NeerajR7
/
PWA-E commerce
Your new repository will be created as PWA-E-commerce.
The repository name can only contain ASCII letters, digits, and the characters ., -, and _.
Great repository names are short and memorable. Need inspiration? How about redesigned-guide ?
Description (optional)
Public
Anyone on the internet can see this repository. You choose who can commit.
Private
You choose who can see and commit to this repository.
Initialize this repository with:
Add a README file
This is where you can write a long description for your project. Learn more about READMEs.
Add .gitignore
.gitignore template: None
Choose which files not to track from a list of templates. Learn more about ignoring files.
Choose a license
License: None





Conclusion : Therefore, deploying an E-commerce PWA to GitHub Pages involves setting up a GitHub repository, configuring GitHub Pages settings, and ensuring proper functionality of the deployed PWA. This deployment method offers a convenient and cost-effective way to host PWAs with minimal setup and maintenance requirements

EXPERIMENT : 11

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

Theory :

What is Lighthouse?
Lighthouse is an open-source tool that audits your web app for various aspects, such as accessibility, performance, best practices, and SEO. You can run Lighthouse in Chrome DevTools, from the command line, or as a Node module. Lighthouse generates a report that gives you scores and recommendations for improving your web app.

How to use Lighthouse for PWAs?

To use Lighthouse for PWAs, you need to open Chrome DevTools and go to the Lighthouse tab, select the PWA option, and click Generate report. After the report is finished, review the results and fix any issues that Lighthouse identifies. Once you have made all of the necessary changes, rerun the audit until you get a high score. This will ensure that your web app meets the criteria for being a progressive web app, such as having a web app manifest, a service worker, responsive design, and more.
Objective: Utilize Google Lighthouse PWA Analysis Tool to evaluate the functionality of a Progressive Web App (PWA).

Key features include:

1. Audit Capabilities: Google Lighthouse conducts audits on different facets of web pages and applications, assessing performance metrics, accessibility standards, coding practices, SEO factors, and PWA criteria.

2. Scoring System: Lighthouse assigns scores ranging from 0 to 100 for each audit category, with higher scores indicating better compliance and performance. Detailed insights and recommendations accompany these scores.

3. Detailed Reports: Following an audit, Lighthouse generates detailed reports highlighting scores, metrics, and actionable recommendations to optimize web pages and applications.

4. Integration with DevTools: Lighthouse seamlessly integrates into Chrome DevTools, facilitating easy access for developers to conduct audits directly within the browser.

5. Open-Source and Extensible: Being open-source, Lighthouse allows for community contributions and customization. It supports plugins and extensions for enhanced functionality.

6. Focus on Performance Optimization: Lighthouse prioritizes performance optimization, aiding developers in identifying and rectifying issues that impact page load times and user experience.

Procedure:
To evaluate the functionality of a Progressive Web App using Google Lighthouse, follow these steps:

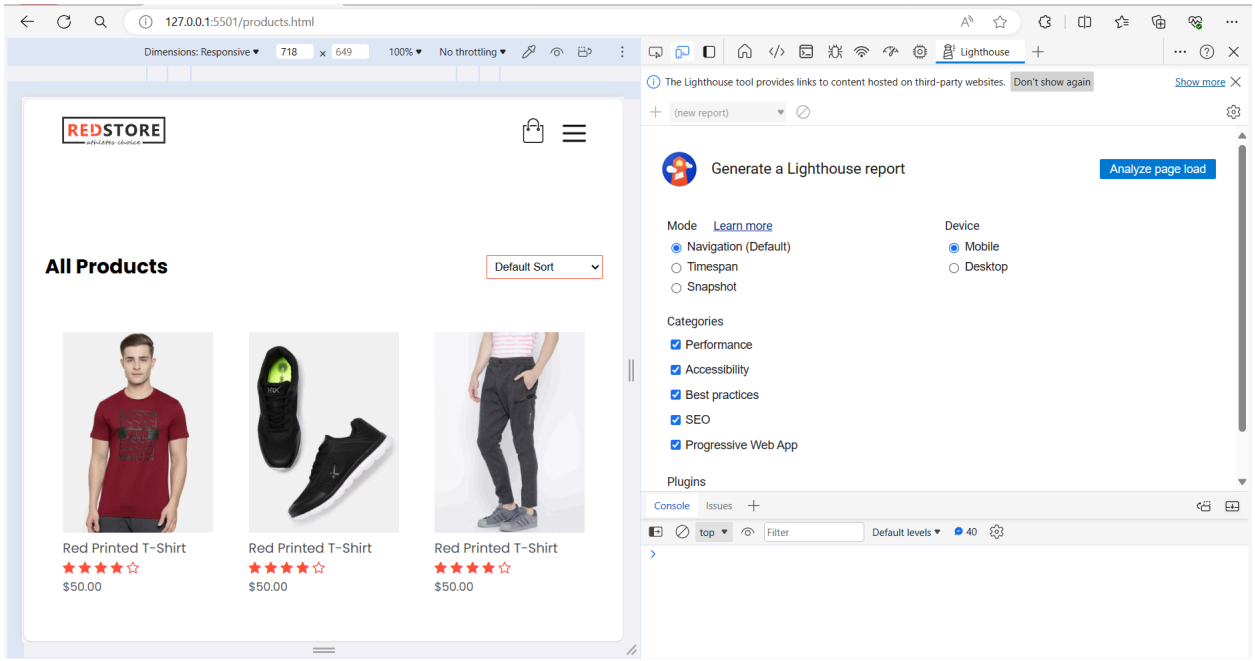
1. Open Chrome DevTools: Launch Google Chrome browser and navigate to the desired website. Right-click on the page and select "Inspect" or use the keyboard shortcut to open Chrome DevTools.

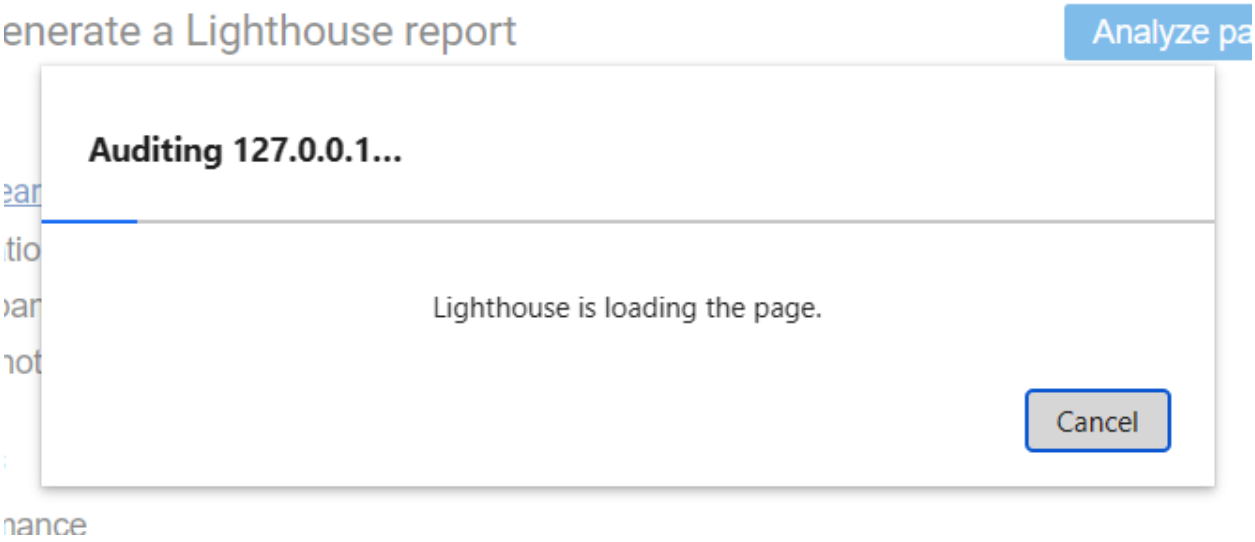
2. Access Lighthouse Tab: Within Chrome DevTools, navigate to the "Lighthouse" tab located at the top of the panel.

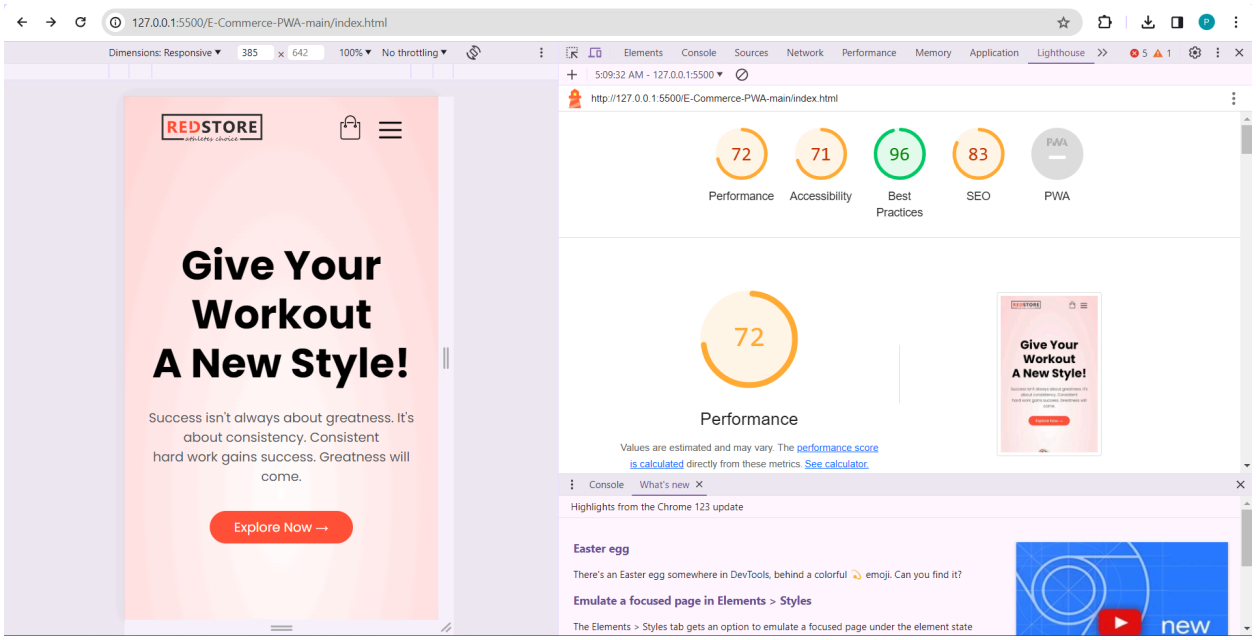
3. Run Lighthouse Audit: Initiate the audit process by clicking on the "Generate report" button. Select the audit categories, including "Progressive Web App," and proceed.

4. Review Audit Results: Once the audit completes, Lighthouse presents a comprehensive report containing scores and detailed insights for each category. Evaluate the PWA section to verify compliance with PWA criteria and receive suggestions for enhancements.

Output:


127.0.0.1:5501/products.html
REDSTORE
All Products
Default Sort
Red Printed T-Shirt
$50.00
Red Printed T-Shirt
$50.00
Red Printed T-Shirt
$50.00
Lighthouse
Generate a Lighthouse report
Analyze page load
Mode
Learn more
Navigation (Default)
Timespan
Snapshot
Device
Mobile
Desktop
Categories
Performance
Accessibility
Best practices
SEO
Progressive Web App
Plugins
Console
Issues


Auditing 127.0.0.1...
Lighthouse is loading the page.
Cancel



Conclusion : We have explored the functionality of Google Lighthouse PWA Analysis Tool and utilized it to assess the performance statistics of our E-commerce Progressive Web Application, Kitter. Through Lighthouse's auditing capabilities, we gained valuable insights to optimize our PWA and enhance user experience.