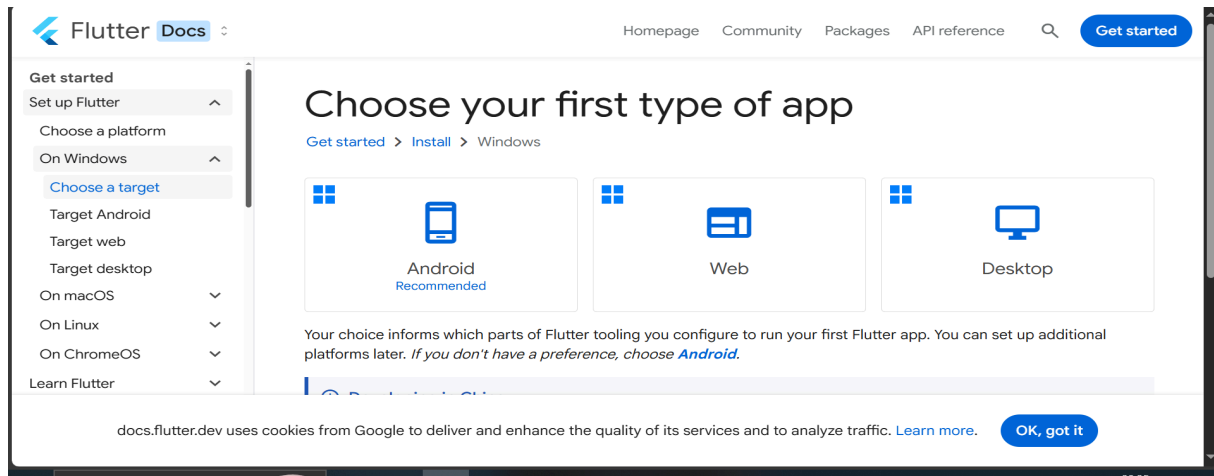


**MAD & PWA Lab
Experiment 01****Aim: To install and configure the Flutter Environment****Theory:**

Flutter is an open-source UI toolkit from Google used for building cross-platform apps. Setting up Flutter involves installing the SDK, configuring environment variables, and setting up an IDE like VS Code or Android Studio.

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.



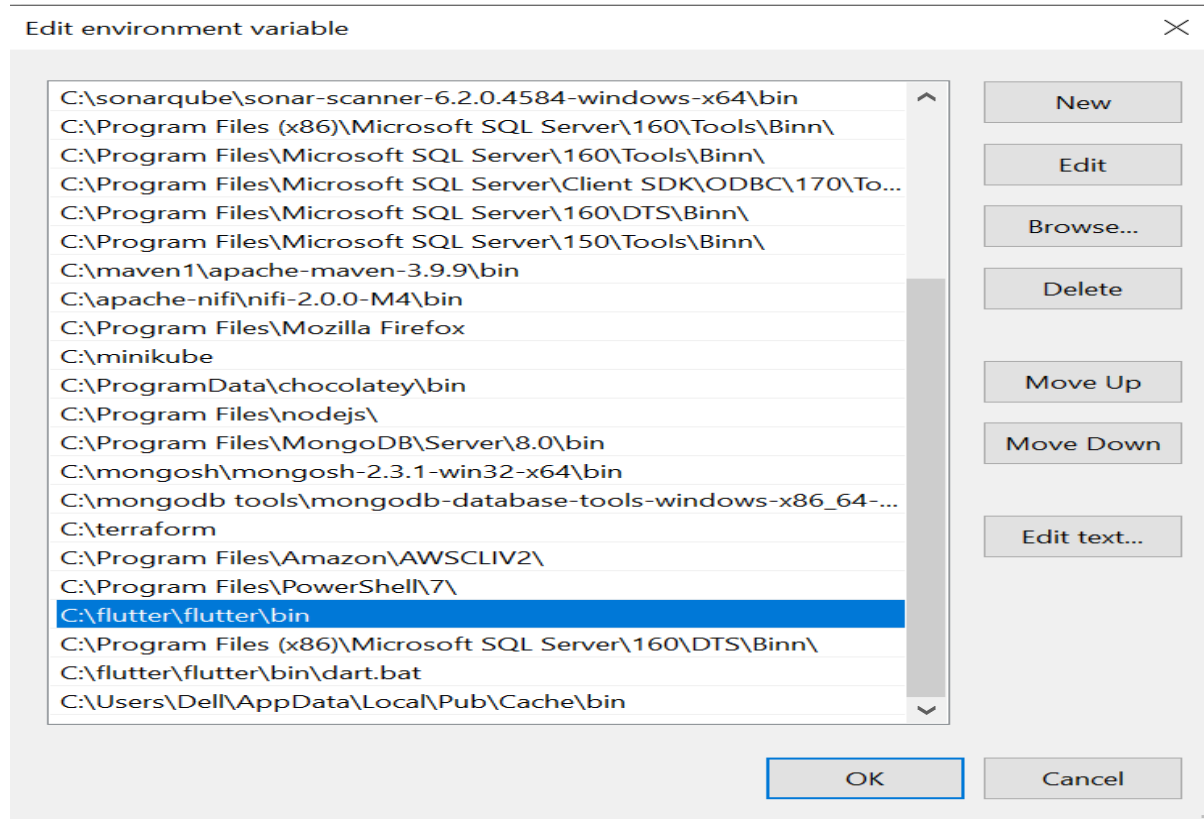
Step 2: Next, to download the latest Flutter SDK, click on the Windows icon. Here, you will find the download link for SDK.

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 command prompt.

```

Command Prompt - flutter
Microsoft Windows [Version 10.0.19045.5679]
(c) Microsoft Corporation. All rights reserved.

C:\Users\Dell>flutter

A new version of Flutter is available!
To update to the latest version, run "flutter upgrade".

Manage your Flutter app development.

Common commands:

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

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

Usage: flutter <command> [arguments]

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

Available commands:

Flutter SDK
bash-completion          Output command line shell completion setup scripts.
channel                  List or switch Flutter channels.
config                   Configure Flutter settings.
doctor                   Show information about the installed tooling.
  
```

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.

```
C:\Users\jalpa>
C:\Users\jalpa>
C:\Users\jalpa>flutter doctor
Running "flutter pub get" in flutter_tools... 17.0s
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 2.8.1, on Microsoft Windows [Version 10.0.19042.1415], locale en-US)
[X] Android toolchain - develop for Android devices
    X Unable to locate Android SDK.
      Install Android Studio from: https://developer.android.com/studio/index.html
      On first launch it will assist you in installing the Android SDK components.
      (or visit https://flutter.dev/docs/get-started/install/windows#android-setup for detailed instructions).
      If the Android SDK has been installed to a custom location, please use
      'flutter config --android-sdk' to update to that location.

[✓] Chrome - develop for the web
[!] Android Studio (not installed)
[✓] VS Code (version 1.55.2)
[✓] Connected device (2 available)

! Doctor found issues in 2 categories.

C:\Users\jalpa>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 2.8.1, on Microsoft Windows [Version 10.0.19042.1415], locale en-US)
[!] Android toolchain - develop for Android devices (Android SDK version 32.0.0)
    X cmdline-tools component is missing
      Run 'path/to/sdkmanager --install "cmdline-tools;latest"'
      See https://developer.android.com/studio/command-line for more details.
    X Android license status unknown.
      Run 'flutter doctor --android-licenses' to accept the SDK licenses.
      See https://flutter.dev/docs/get-started/install/windows#android-setup for more details.
[✓] Chrome - develop for the web
[✓] Android Studio (version 2020.3)
[✓] VS Code (version 1.55.2)
[✓] Connected device (2 available)

! Doctor found issues in 1 category.
```

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.



Conclusion: A properly configured Flutter environment enables smooth development, testing, and deployment of apps like the Society Profile app.

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

Flutter provides a rich set of widgets such as Text, Container, Row, Column, ListView, etc., that allow flexible UI design. These widgets form the building blocks of any Flutter interface.

Sample code for the registration page containing common widgets is as follows:

```
import 'package:flutter/material.dart';
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'society_list_screen.dart';

class RegisterScreen extends StatefulWidget {
  final User user;

  const RegisterScreen({Key? key, required this.user}) : super(key: key);

  @override
  _RegisterScreenState createState() => _RegisterScreenState();
}

class _RegisterScreenState extends State<RegisterScreen> {
  final FirebaseFirestore _db = FirebaseFirestore.instance;
  final TextEditingController _contactController = TextEditingController();
  final TextEditingController _societyController = TextEditingController();
  final TextEditingController _flatNumberController = TextEditingController();
  final _formKey = GlobalKey<FormState>();

  String _userType = "Resident"; // Default user type
  bool _isResident = true;
  bool _isLoading = false;

  Future<void> _registerUser() async {
    if (!_formKey.currentState!.validate()) return;
    setState(() => _isLoading = true);

    try {
      String societyId = ""; // Initialize society ID

      // If user is a Manager, create a society
      if (_userType == "Manager") {
        DocumentReference societyRef = await _db.collection("societies").add({
          "name": _societyController.text.trim(),
          "location": "Not specified", // Default location, can be updated later
          "createdBy": widget.user.uid, // Store Manager's UID
        });
        societyId = societyRef.id; // Save society ID
      }

      // Store user details in Firestore
    }
  }
}
```

```
await _db.collection("users").doc(widget.user.uid).set({
  "uid": widget.user.uid,
  "name": widget.user.displayName ?? "Unknown",
  "email": widget.user.email,
  "contactNumber": _contactController.text.trim(),
  "userType": _userType,
  "societyName": _societyController.text.trim(),
  "societyId": _userType == "Manager" ? societyId : "", // Assign society to Managers only
  "flatNumber": _isResident ? _flatNumberController.text.trim() : "N/A",
  "photoUrl": widget.user.photoURL ?? "",
  "createdAt": FieldValue.serverTimestamp(),
});

Navigator.pushReplacement(
context,
MaterialPageRoute(
  builder: (context) => SocietyListScreen(
    userId: widget.user.uid,
    userType: _userType, // Ensure correct value is passed
  ),
),
);

ScaffoldMessenger.of(context).showSnackBar(
  const SnackBar(content: Text("✅ Registration Successful!")),
);
} catch (e) {
  ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(content: Text("❌ Error: ${e.toString()}")),
  );
} finally {
  setState(() => _isLoading = false);
}
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text("Register")),
    body: GestureDetector(
      onTap: () => FocusScope.of(context).unfocus(),
      child: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Form(
          key: _formKey,
          child: SingleChildScrollView(
            child: Column(
              crossAxisAlignment: CrossAxisAlignment.start,
              children: [
```

```
const Text(
  "Complete Your Registration",
  style: TextStyle(fontSize: 22, fontWeight: FontWeight.bold),
),
const SizedBox(height: 20),

// Full Name
TextFormField(
  initialValue: widget.user.displayName ?? "Unknown",
  readOnly: true,
  decoration: const InputDecoration(
    labelText: "Full Name",
    border: OutlineInputBorder(),
  ),
),
const SizedBox(height: 10),

// Email
TextFormField(
  initialValue: widget.user.email,
  readOnly: true,
  decoration: const InputDecoration(
    labelText: "Email",
    border: OutlineInputBorder(),
  ),
),
const SizedBox(height: 10),

// Contact Number
TextFormField(
  controller: _contactController,
  keyboardType: TextInputType.phone,
  decoration: const InputDecoration(
    labelText: "Contact Number",
    border: OutlineInputBorder(),
  ),
  maxLength: 10,
  validator: (value) {
    if (value!.isEmpty) return "Enter Contact Number";
    // if (!RegExp(r'^[0-9]{10}$').hasMatch(value)) return "Enter a valid 10-digit
number";
    return null;
  },
),
const SizedBox(height: 10),

// User Type Dropdown
DropdownButtonFormField<String>(
  value: _userType,
  items: ["Resident", "Manager"].map((type) {
    return DropdownMenuItem(value: type, child: Text(type));
```

```

    }).toList(),
    onChanged: (value) {
      setState(() {
        _userType = value!;
        _isResident = _userType == "Resident";
      });
    },
    decoration: const InputDecoration(
      labelText: "User Type",
      border: OutlineInputBorder(),
    ),
  ),
  const SizedBox(height: 10),

  // Society Name
  TextFormField(
    controller: _societyController,
    decoration: const InputDecoration(
      labelText: "Society Name",
      border: OutlineInputBorder(),
    ),
    validator: (value) => value!.isEmpty ? "Enter Society Name" : null,
  ),
  const SizedBox(height: 10),

  // Flat Number (Only for Residents)
  if (_isResident)
    TextFormField(
      controller: _flatNumberController,
      decoration: const InputDecoration(
        labelText: "Flat Number",
        border: OutlineInputBorder(),
      ),
      validator: (value) => value!.isEmpty ? "Enter Flat Number" : null,
    ),
  const SizedBox(height: 20),

  // Register Button
  SizedBox(
    width: double.infinity,
    child: ElevatedButton(
      onPressed: _isLoading ? null : _registerUser,
      child: _isLoading
        ? const CircularProgressIndicator(color: Colors.white)
        : const Text("Complete Registration"),
    ),
  ),
],
),
),
),
),

```



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

In the SignUp form, multiple commonly used Flutter widgets have been utilized to create an interactive and well-structured UI:

- **TextField**: Used to capture user input such as email, name, and flat number. These are interactive fields allowing text input.
- **DropDownButtonFormField**: Used to allow users to select options from a dropdown list — for example, selecting "Resident" or "Manager" as the user type.
- **Column**: Used to align widgets vertically — making sure the form fields are placed one below the other in a clean stacked layout.
- **Row**: Used wherever horizontal alignment is required, such as displaying form elements side-by-side or organizing parts of the layout.
- **Padding and SizedBox**: Used for spacing and improving layout aesthetics — giving proper margin and breathing room between widgets.
- **ElevatedButton**: A clickable button that triggers actions, such as submitting the registration form.
- **Form and GlobalKey<FormState>**: While not visual widgets, they are crucial for validating the form and maintaining form state.
- **Responsive Design Considerations**: The use of flexible layouts and widgets ensures that the form looks good on different screen sizes.



Register

Complete Your Registration

Full Name
Mohit Kerkar

Email
ironcaptain007@gmail.com

Contact Number

User Type
Resident

Society Name

Flat Number

Complete Registration

Conclusion: Understanding and utilizing core widgets allowed us to build a structured, user-friendly Society Profile UI.

Experiment 03**Aim: To include icons, Prerequisite, fonts in Flutter app****Theory:**

In Flutter, visual consistency and user engagement are enhanced by incorporating icons and custom fonts into an application. These elements contribute to the overall branding and aesthetic appeal of the app.

To include such assets, the pubspec.yaml file acts as the central configuration hub.

Required Flutter packages such as cupertino_icons must be specified in the dependencies section of the pubspec.yaml.

Assets like fonts or icon files must be placed in appropriate folders within the project directory.

```
dependencies:  
  
  cupertino_icons: ^1.0.8
```

- This line ensures that Cupertino icons are available, enabling the use of iOS-style icons throughout the app.
- uses-material-design: true under the flutter: section enables the use of Google's Material Icons in the app.

```
flutter:  
  
  uses-material-design: true
```

To include fonts

```
flutter:  
  uses-material-design: true  
  fonts:  
    - family: Roboto  
      fonts:  
        - asset: assets/fonts/Roboto-Regular.ttf
```

Conclusion: Custom icons and fonts improved the identity and user experience of the Society Profile app.

Experiment 04**Aim: To create an interactive Form using form widget****Theory:**

Flutter forms use Form, TextFormField, and GlobalKey<FormState> for validation and user input handling. They are essential for collecting structured user data.

Reference

1. <https://docs.flutter.dev/cookbook/forms/validation>
2. <https://www.javatpoint.com/flutter-forms>
3. Example <https://codelabs.developers.google.com/codelabs/first-flutter-app-pt2#6>
4. <https://flutterbyexample.com/lesson/stateful-widget-lifecycle>

Steps

- Create a Form with a GlobalKey
- Add a TextFormField with validation logic
- Create a button to validate and submit the form
- I am using a Form widget with GlobalKey<FormState>.
- Validation, conditional fields (e.g., flat number for residents), and user input are handled properly.

Source code is as follows:

```
import 'package:flutter/material.dart';
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'society_list_screen.dart';

class RegisterScreen extends StatefulWidget {
  final User user;

  const RegisterScreen({Key? key, required this.user}) : super(key: key);

  @override
  _RegisterScreenState createState() => _RegisterScreenState();
}

class _RegisterScreenState extends State<RegisterScreen> {
  final FirebaseFirestore _db = FirebaseFirestore.instance;
  final TextEditingController _contactController = TextEditingController();
  final TextEditingController _societyController = TextEditingController();
  final TextEditingController _flatNumberController = TextEditingController();
  final _formKey = GlobalKey<FormState>();

  String _userType = "Resident"; // Default user type
  bool _isResident = true;
  bool _isLoading = false;

  Future<void> _registerUser() async {
    if (!_formKey.currentState!.validate()) return;
    setState(() => _isLoading = true);
```

```
try {
  String societyId = ""; // Initialize society ID

  // If user is a Manager, create a society
  if (_userType == "Manager") {
    DocumentReference societyRef = await _db.collection("societies").add({
      "name": _societyController.text.trim(),
      "location": "Not specified", // Default location, can be updated later
      "createdBy": widget.user.uid, // Store Manager's UID
    });
    societyId = societyRef.id; // Save society ID
  }

  // Store user details in Firestore
  await _db.collection("users").doc(widget.user.uid).set({
    "uid": widget.user.uid,
    "name": widget.user.displayName ?? "Unknown",
    "email": widget.user.email,
    "contactNumber": _contactController.text.trim(),
    "userType": _userType,
    "societyName": _societyController.text.trim(),
    "societyId": _userType == "Manager" ? societyId : "", // Assign society to Managers only
    "flatNumber": _isResident ? _flatNumberController.text.trim() : "N/A",
    "photoUrl": widget.user.photoURL ?? "",
    "createdAt": FieldValue.serverTimestamp(),
  });

  Navigator.pushReplacement(
    context,
    MaterialPageRoute(
      builder: (context) => SocietyListScreen(
        userId: widget.user.uid,
        userType: _userType, // Ensure correct value is passed
      ),
    ),
  );

  ScaffoldMessenger.of(context).showSnackBar(
    const SnackBar(content: Text("✅ Registration Successful!")),
  );
} catch (e) {
  ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(content: Text("❌ Error: ${e.toString()}")),
  );
} finally {
  setState(() => _isLoading = false);
}
}
```

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(title: const Text("Register")),
    body: GestureDetector(
      onTap: () => FocusScope.of(context).unfocus(),
      child: Padding(
        padding: const EdgeInsets.all(16.0),
        child: Form(
          key: _formKey,
          child: SingleChildScrollView(
            child: Column(
              crossAxisAlignment: CrossAxisAlignment.start,
              children: [
                const Text(
                  "Complete Your Registration",
                  style: TextStyle(fontSize: 22, fontWeight: FontWeight.bold),
                ),
                const SizedBox(height: 20),

                // Full Name
                TextFormField(
                  initialValue: widget.user.displayName ?? "Unknown",
                  readOnly: true,
                  decoration: const InputDecoration(
                    labelText: "Full Name",
                    border: OutlineInputBorder(),
                  ),
                ),
                const SizedBox(height: 10),

                // Email
                TextFormField(
                  initialValue: widget.user.email,
                  readOnly: true,
                  decoration: const InputDecoration(
                    labelText: "Email",
                    border: OutlineInputBorder(),
                  ),
                ),
                const SizedBox(height: 10),

                // Contact Number
                TextFormField(
                  controller: _contactController,
                  keyboardType: TextInputType.phone,
                  decoration: const InputDecoration(
                    labelText: "Contact Number",
                    border: OutlineInputBorder(),
                  ),
                ),
```

```
        maxLength: 10,
        validator: (value) {
          if (value!.isEmpty) return "Enter Contact Number";
          // if (!RegExp(r'^[0-9]{10}$').hasMatch(value)) return "Enter a valid 10-digit
number";
          return null;
        },
      ),
      const SizedBox(height: 10),

      // User Type Dropdown
      DropdownButtonFormField<String>(
        value: _userType,
        items: ["Resident", "Manager"].map((type) {
          return DropdownMenuItem(value: type, child: Text(type));
        }).toList(),
        onChanged: (value) {
          setState(() {
            _userType = value!;
            _isResident = _userType == "Resident";
          });
        },
        decoration: const InputDecoration(
          labelText: "User Type",
          border: OutlineInputBorder(),
        ),
      ),
      const SizedBox(height: 10),

      // Society Name
      TextFormField(
        controller: _societyController,
        decoration: const InputDecoration(
          labelText: "Society Name",
          border: OutlineInputBorder(),
        ),
        validator: (value) => value!.isEmpty ? "Enter Society Name" : null,
      ),
      const SizedBox(height: 10),

      // Flat Number (Only for Residents)
      if (_isResident)
        TextFormField(
          controller: _flatNumberController,
          decoration: const InputDecoration(
            labelText: "Flat Number",
            border: OutlineInputBorder(),
          ),
          validator: (value) => value!.isEmpty ? "Enter Flat Number" : null,
        ),
      const SizedBox(height: 20),
```

```
        // Register Button
        SizedBox(
          width: double.infinity,
          child: ElevatedButton(
            onPressed: _isLoading ? null : _registerUser,
            child: _isLoading
              ? const CircularProgressIndicator(color: Colors.white)
              : const Text("Complete Registration"),
          ),
        ),
      ],
    ),
  ),
),
);
}
```

In registration form:

- A Form widget is wrapped around the entire input structure to maintain a unified validation context.
- A GlobalKey<FormState> is used to validate and manage form state globally when the form is submitted.
- Various TextFormField widgets are used to collect user inputs like name, email, society, etc.
- Conditional rendering is implemented: for instance, the Flat Number field appears only if the selected user type is 'Resident'. This is achieved using Flutter's stateful widget logic and conditional checks.
- Validation logic is applied to fields to ensure correctness of data (e.g., checking if email is valid, or mandatory fields are not left blank).

Register

Complete Your Registration

Full Name

Mohit Kerkar

Email

ironcaptain007@gmail.com

Contact Number

User Type

Resident

0/10

Society Name

Flat Number

Complete Registration

Conclusion:

We built a dynamic, validated registration form that efficiently captures resident and manager details in the Society app.

Experiment 05**Aim: To apply navigation, routing and gestures in Flutter App****Theory:**

Flutter uses Navigator for route management and gesture widgets like GestureDetector for user interaction. This provides seamless screen transitions and intuitive UX.

- Navigation to HomePage after successful registration.
- You might also have gestures like onTap or onPressed, though not clearly visible here.

Navigation & Routing:

Navigation and routing in Flutter typically involve the use of Navigator.push() or Navigator.pushReplacement() to move between screens.

1. Navigating to Landing Screen (on logout):

```
Navigator.pushReplacement(  
  context,  
  MaterialPageRoute(builder: (context) => LandingScreen()),  
);
```

- Type: pushReplacement (replaces the current route)
- Purpose: Navigates to LandingScreen after logging out

2. Navigating to Profile List Screen (on tapping a society):

```
Navigator.push(  
  context,  
  MaterialPageRoute(  
    builder: (context) => ProfileListScreen(  
      userId: widget.userId,  
      userType: widget.userType,  
      societyId: societyId,  
    ),  
  ),  
);
```

- Type: push (adds a new route to the stack)
- Purpose: Opens the ProfileListScreen with relevant data for the selected society

Gesture Handling:

Gesture handling refers to responding to user interactions like tapping.

1. Tapping a List Item (Society Card):

```
onTap: () {  
  Navigator.push(  
    context,  
    MaterialPageRoute(  
      builder: (context) => ProfileListScreen(  
        userId: widget.userId,  
        userType: widget.userType,  
        societyId: societyId,  
      ),  
    ),  
  );  
}
```

- Gesture: onTap of a ListTile
- Effect: Triggers navigation to a detailed profile list of the selected society

2. Tapping Floating Action Button to Add Society:

```
floatingActionButton: widget.userType == "Manager"  
  ? FloatingActionButton(  
    onPressed: _addSociety,  
    ...  
  )  
  : null,
```

- Gesture: Tap on the FAB
- Effect: Opens a dialog to add a new society (managers only)

3. Tapping the "Add" Button in Dialog:

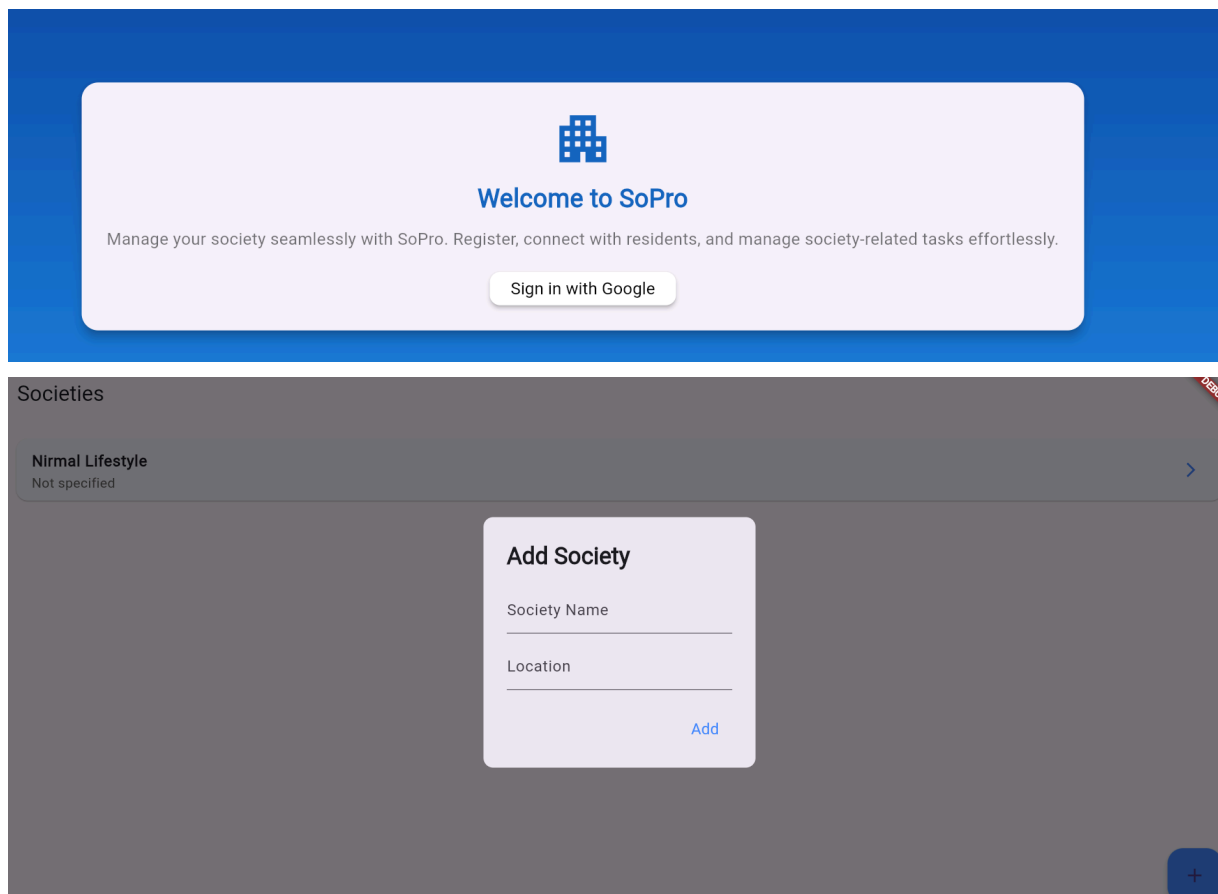
```
TextButton(  
  onPressed: () async {  
    ...  
    Navigator.pop(context);  
  },  
  child: Text("Add"),  
)
```

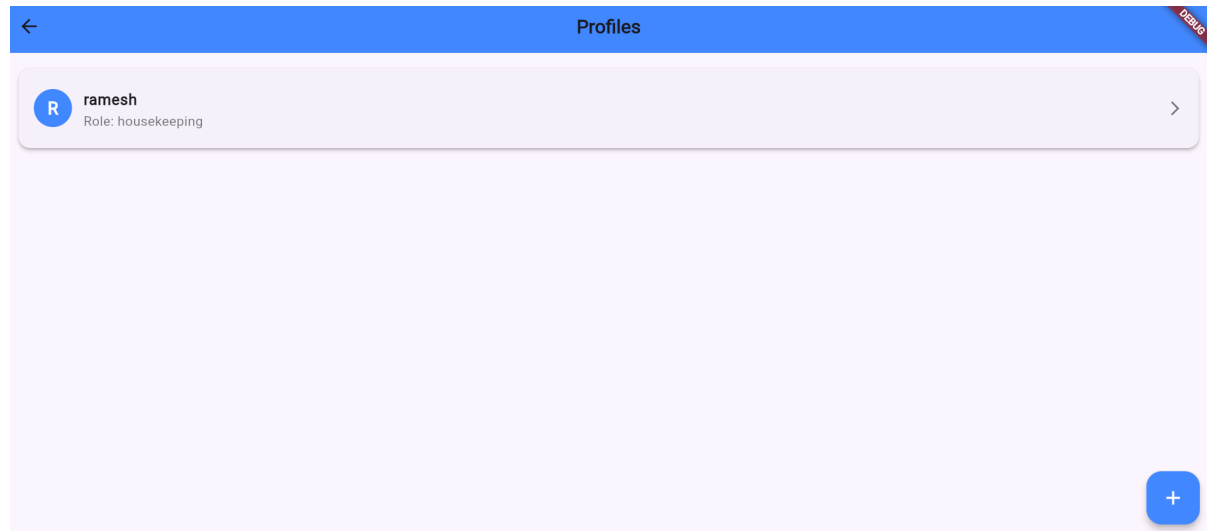
- Gesture: Tap on the TextButton
- Effect: Adds the society to Firestore and closes the dialog

Summary:

Type	Location in Code	Purpose
Navigation	Navigator.pushReplacement(...)	Redirects to LandingScreen after logout
Navigation	Navigator.push(...)	Opens ProfileListScreen when a society is tapped
Gesture	onTap in ListTile	Handles society list item selection
Gesture	onPressed of FloatingActionButton	Opens dialog to add a new society
Gesture	onPressed of TextButton inside AlertDialog	Adds the new society and dismisses dialog

Let me know if you want a visual flow of the navigation or want this section commented inline in your code!





Conclusion:

Effective routing and gestures enabled smooth multi-page transitions within the Society Management platform.

Experiment 06**Aim: To Connect Flutter UI with fireBase database****Theory:**

Firestore offers a backend-as-a-service including authentication and Firestore database. Integration allows storing, updating, and retrieving user data in real-time.

- Firebase Auth is used for sign-in.
- Firestore is used to store user data (users collection) and societies.

References:

https://youtu.be/ErP_xomHKTW

<https://www.digitalocean.com/community/tutorials/flutter-firebase-setup>

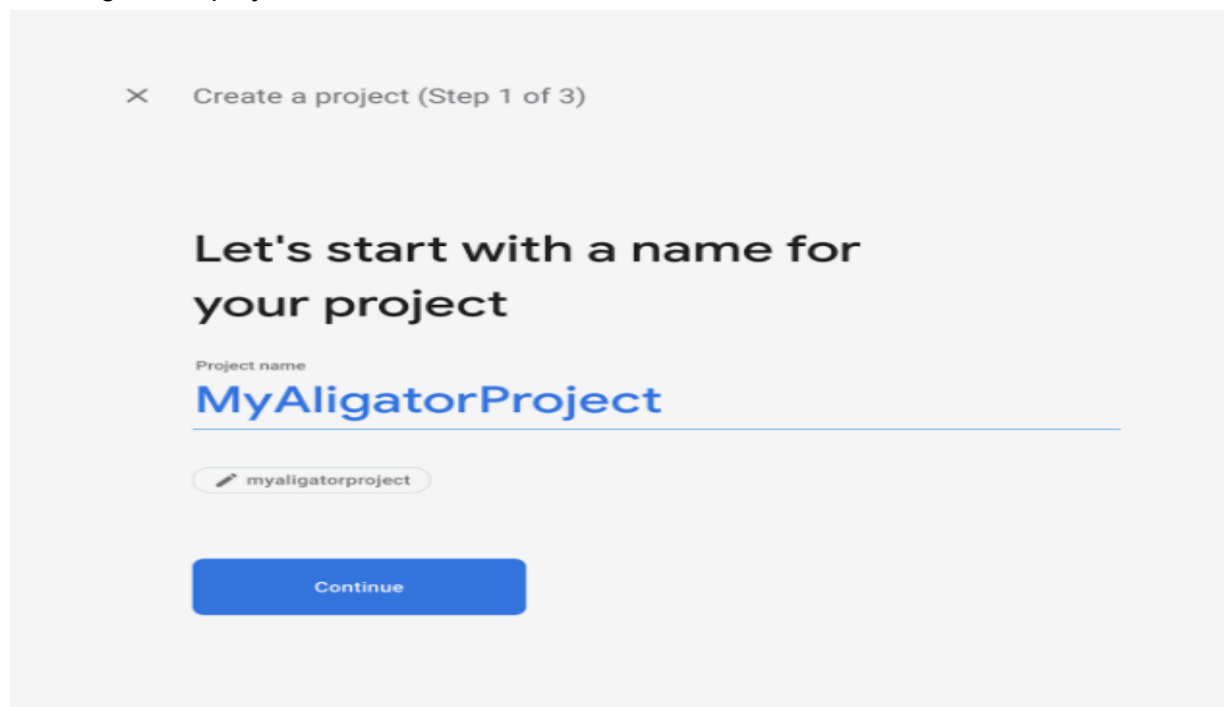
Prerequisites

To complete this tutorial, you will need:

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

This tutorial was verified with Flutter v2.0.6, Android SDK v31.0.2, and Android Studio v4.1.

Creating a new project







✕ Create a project (Step 2 of 3)



Google Analytics for your Firebase project



Google Analytics is a free and unlimited analytics solution that enables targeting, reporting, and more in Firebase Crashlytics, Cloud Messaging, In-App Messaging, Remote Config, A/B Testing, Predictions, and Cloud Functions.



Google Analytics enables:



 A/B testing 

 User segmentation & targeting across
Firebase products 

 Predicting user behavior 

 Crash-free users 

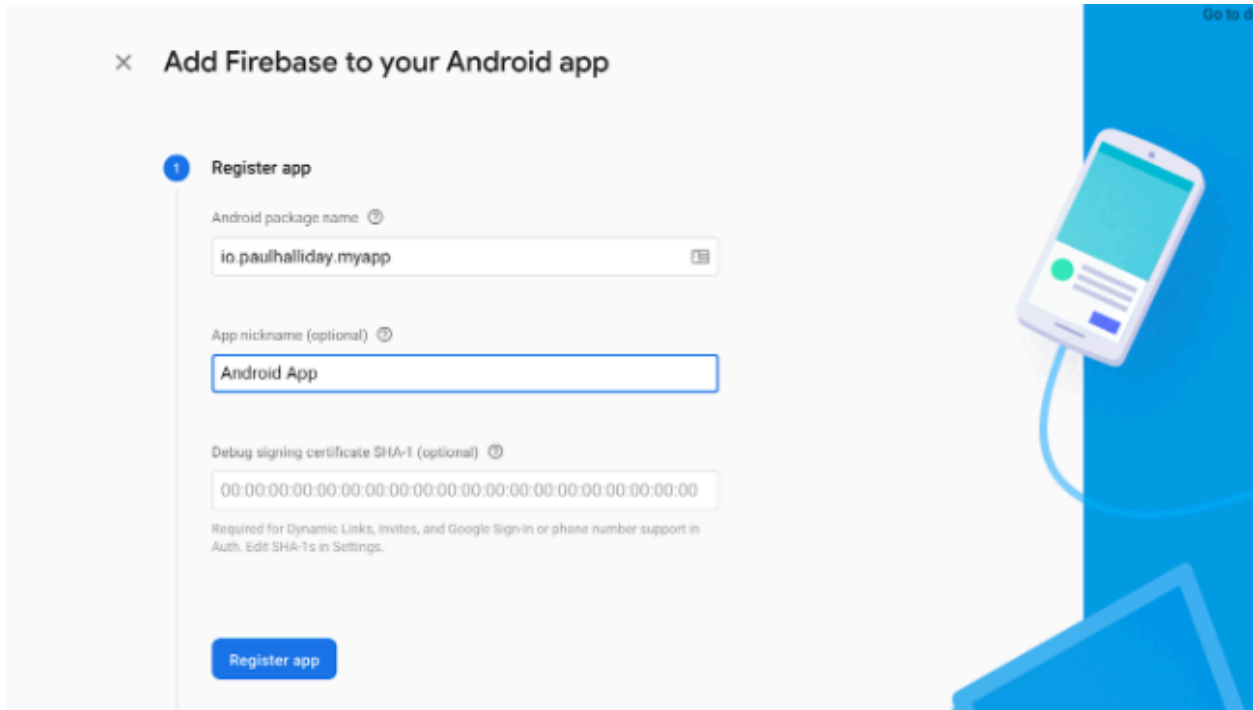
 Event-based Cloud Functions triggers 

 Free unlimited reporting 

☒ Enable Google Analytics for this project
Recommended

[Previous](#)

[Continue](#)



× Add Firebase to your Android app

1 Register app

Android package name ⓘ

io.paulhalliday.myapplication ⓘ

App nickname (optional) ⓘ

Android App

Debug signing certificate SHA-1 (optional) ⓘ

00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00:00

Required for Dynamic Links, invites, and Google Sign-in or phone number support in Auth. Edit SHA-1s in Settings.

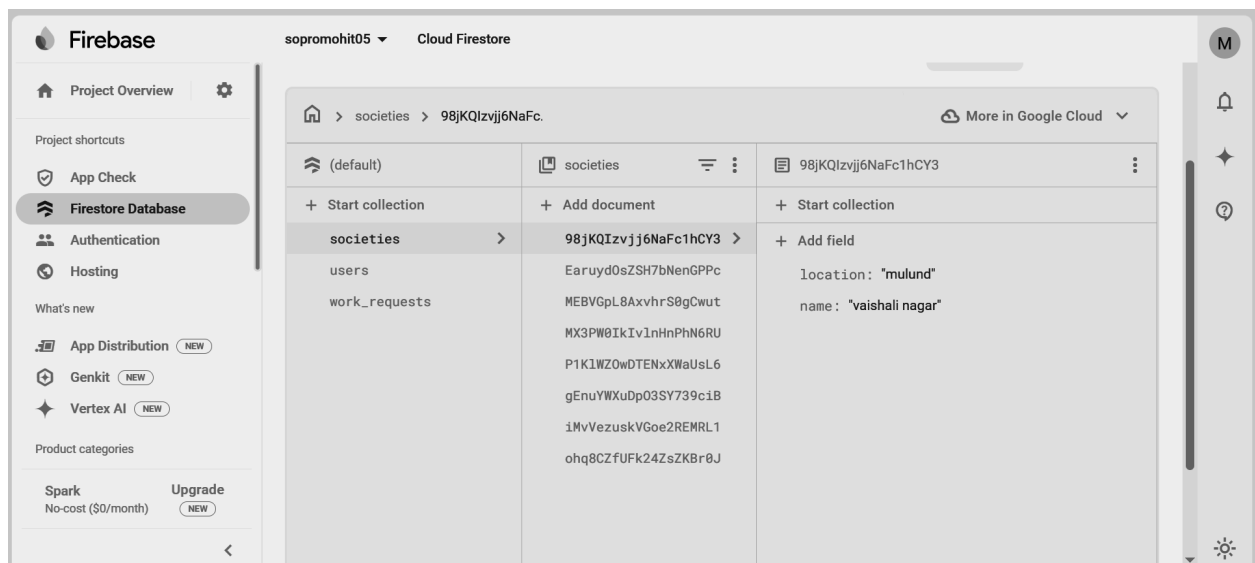
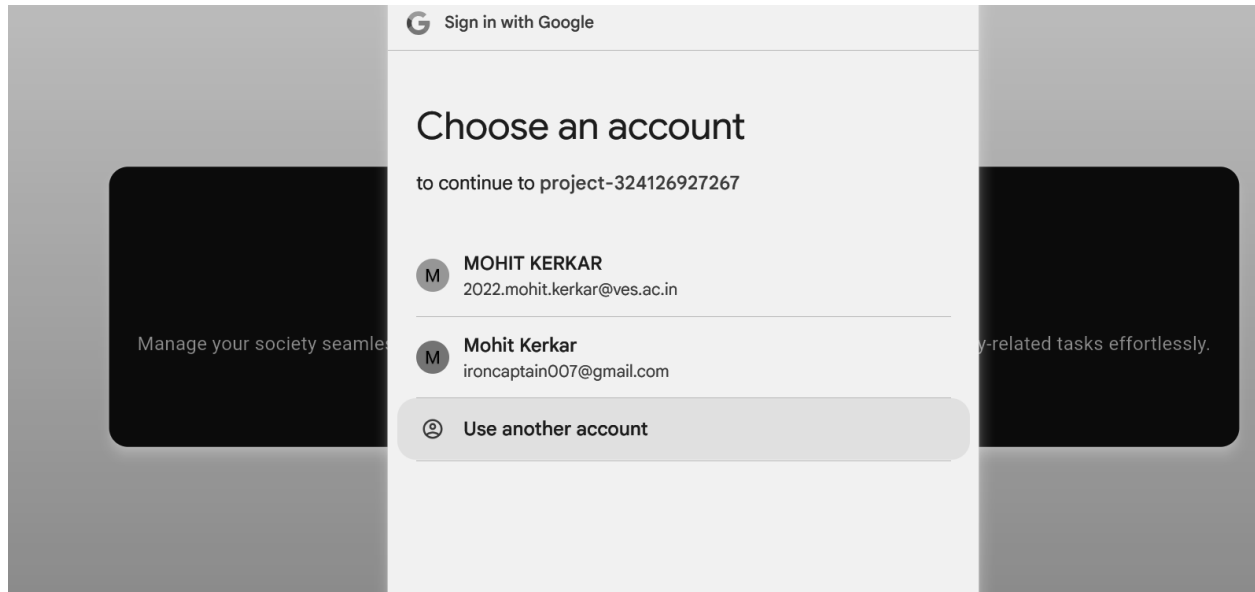
Register app

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

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

```
com.example.flutterfirebaseexample
```

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



Conclusion:

Firebase integration made user registration, profile management, and data sync real-time and secure.

Experiment No. 7

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

Online Reference:

<https://developer.mozilla.org/en-US/docs/Web/Manifest>

<https://www.geeksforgeeks.org/making-a-simple-pwa-under-5-minutes/>

Theory:-

Regular Web App

A regular web app is a website that is designed to be accessible on all mobile devices such that the content gets fit as per the device screen. It is designed using a web technology stack (HTML, CSS, JavaScript, Ruby, etc.) and operates via a browser. They offer various native-device features and functionalities. However, it entirely depends on the browser the user is using. In other words, it might be possible that you can access a native-device feature on Chrome but not on Safari or Mozilla Firefox because the browsers are incompatible with that feature.

Progressive Web App

Progressive Web App (PWA) is a regular web app, but some extras enable it to deliver an excellent user experience. It is a perfect blend of desktop and mobile application experience to give both platforms to the end-users.

Difference between PWAs vs. Regular Web Apps:

A Progressive Web is different and better than a Regular Web app with features like:

1. Native Experience

Though a PWA runs on web technologies (HTML, CSS, JavaScript) like a Regular web app, it gives user experience like a native mobile application. It can use most native device features, including push notifications, without relying on the browser or any other entity. It offers a seamless and integrated user experience that it is quite tough for one to differentiate between a PWA and a Native application by considering its look and feel.

2. Ease of Access

Unlike other mobile apps, PWAs do not demand longer download time and make memory space available for installing the applications. The PWAs can be shared and installed by a link, which cuts down the number of steps to install and use. These applications can easily keep an app icon on the user's home screen, making the app easily accessible to the users and helps the brands remain in the users' minds, and improving the chances of interaction.

3. Faster Services

PWAs can cache the data and serve the user with text stylesheets, images, and other web content even before the page loads completely. This lowers the waiting time for the end-users and helps the brands improve the user engagement and retention rate, which eventually adds value to their business.

4. Engaging Approach

As already shared, the PWAs can employ push notifications and other native device features more efficiently. Their interaction does not depend on the browser user uses. This eventually improves the chances of notifying the user regarding your services, offers, and other options related to your brand and keeping them hooked to your brand. In simpler words, PWAs let you maintain the user engagement and retention rate.

5. Updated Real-Time Data Access

Another plus point of PWAs is that these apps get updated on their own. They do not demand the end-users to go to the App Store or other such platforms to download the update and wait until installed.

In this app type, the web app developers can push the live update from the server, which reaches the apps residing on the user's devices automatically. Therefore, it is easier for the mobile app developer to provide the best of the updated functionalities and services to the end-users without forcing them to update their app.

6. Discoverable

PWAs reside in web browsers. This implies higher chances of optimizing them as per the Search Engine Optimization (SEO) criteria and improving the Google rankings like that in websites and other web apps.

7. Lower Development Cost

Progressive web apps can be installed on the user device like a native device, but it does not demand submission on an App Store. This makes it far more cost-effective than native mobile applications while offering the same set of functionalities.

Pros and cons of the Progressive Web App

The main features are:

Progressive — They work for every user, regardless of the browser chosen because they are built at the base with progressive improvement principles.

Responsive — They adapt to the various screen sizes: desktop, mobile, tablet, or dimensions that can later become available.

App-like — They behave with the user as if they were native apps, in terms of interaction and navigation.

Updated — Information is always up-to-date thanks to the data update process offered by service workers.

Secure — Exposed over HTTPS protocol to prevent the connection from displaying information or altering the contents.

Searchable — They are identified as “applications” and are indexed by search engines.

Reactivable — Make it easy to reactivate the application thanks to capabilities such as web notifications.

Installable — They allow the user to “save” the apps that he considers most useful with the corresponding icon on the screen of his mobile terminal (home screen) without having to face all the steps and problems related to the use of the app store.

Linkable — Easily shared via URL without complex installations.

Offline — Once more it is about putting the user before everything, avoiding the usual error message in case of weak or no connection. The PWA are based on two particularities: first of all the ‘skeleton’ of the app, which recalls the page structure, even if its contents do not respond and its elements include the header, the page layout, as well as an illustration that signals that the page is loading.

Weaknesses refer to:

IOS support from version 11.3 onwards;

Greater use of the device battery;

Not all devices support the full range of PWA features (same speech for iOS and Android operating systems);

It is not possible to establish a strong re-engagement for iOS users (URL scheme, standard web notifications);

Support for offline execution is however limited;

Lack of presence on the stores (there is no possibility to acquire traffic from that channel);

There is no “body” of control (like the stores) and an approval process;

Limited access to some hardware components of the devices;

Little flexibility regarding “special” content for users (eg loyalty programs, loyalty, etc.).

Sample code:

```
{
{
  "name": "ShopEase - Your Ecommerce Store",
  "short_name": "ShopEase",
  "start_url": "/index.html",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#4CAF50",
  "icons": [
    {
      "src": "icons/icon-192x192.png",
      "sizes": "192x192",
      "type": "image/png"
    },
    {
      "src": "icons/icon-512x512.png",
```

```
    "sizes": "512x512",  
    "type": "image/png"  
  }  
]  
}
```

Welcome to ShopEase!

Your favorite online store as a PWA.



Try adding this app to your home screen!



Conclusion:

Adding a manifest file helped transform our app into a PWA with installable and home screen access support.

Experiment 08

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:

A service worker is a background script that intercepts network requests. Installing and activating it is the first step toward enabling offline features and caching in a PWA.

Service Worker in Progressive Web Apps (PWAs)

A Service Worker is a script that the browser runs in the background, separate from a web page, enabling features such as offline access, background synchronization, and push notifications. It is a core technology in building Progressive Web Apps (PWAs) that function reliably regardless of network conditions.

1. What is a Service Worker?

A Service Worker acts as a network proxy between the web application and the internet. It intercepts network requests, giving developers the ability to:

- Control caching strategies,
- Pre-fetch and cache assets,
- Serve content from the cache when offline.

2. Service Worker Lifecycle

The service worker lifecycle includes three main phases:

- Registration
The service worker is linked to the browser via the main JavaScript file.
- Installation
During this phase, static assets such as HTML, CSS, JavaScript, and images are cached.
- Activation
The service worker takes control of all pages under its scope and cleans up any old caches if necessary.

3. Registration Process

To register a service worker, include the following code in your main JavaScript file:

```
if ('serviceWorker' in navigator) {  
  window.addEventListener('load', () => {  
    navigator.serviceWorker.register('/service-worker.js')  
      .then(registration => {  
        console.log('Service Worker registered with scope:', registration.scope);  
      })  
      .catch(error => {  
        console.log('Service Worker registration failed:', error);  
      });  
  });  
}
```

Once registered, the browser manages the lifecycle automatically, and the service worker is activated if all steps succeed.

#	Name	Response-Ty..	Content-Type	Content-Len..	Time Cached	Vary Header
33	/google/callback?state=1QwhAYXqBtgreiQmMhMGF3dUdliu...	basic	text/html; ch...	2,393	3/20/2022, 1...	Accept-Enco...
34	/google/callback?state=saf6d7IdKswSCSaD1jBYUvjdMhifCotf...	basic	text/html; ch...	2,393	3/20/2022, 1...	Accept-Enco...
35	/google/callback?state=tIN8tMp0M51GoSrV50higxVih8geYc...	basic	text/html; ch...	2,393	3/20/2022, 1...	Accept-Enco...
36	/google/callback?state=y1AGIXLn8r1ggjlojggpYSRpVV1osGod...	basic	text/html; ch...	2,393	3/20/2022, 1...	Accept-Enco...
37	/images/overview.jpg	basic	text/html; ch...	418	3/20/2022, 1...	Accept-Enco...
38	/ios/144.png	basic	image/png	12,203	3/20/2022, 1...	Accept-Enco...
39	/js/offcanvas.js	basic	application/x...	223	3/23/2022, 1...	Accept-Enco...
40	/js/share.js	basic	application/x...	276	3/23/2022, 1...	Accept-Enco...
41	/manifest.json	basic	application/js...	1,076	3/23/2022, 1...	Accept-Enco...
42	/menu	basic	text/html; ch...	5,981	3/23/2022, 1...	Accept-Enco...
43	/offline.html	basic	text/html	662	3/20/2022, 1...	Accept-Enco...

Conclusion:

The install and activate phases successfully set up our Society App's offline-ready PWA architecture.

MAD and PWA Lab
EXPERIMENT NO. 9

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

Theory:**Service Worker**

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

Things to note about Service Worker:

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

Fetch Event

You can track and manage page network traffic with this event. You can check existing cache, manage “cache first” and “network first” requests and return a response that you want.

Of course, you can use many different methods but you can find in the following example a “cache first” and “network first” approach. In this example, if the request's and current location's origin are the same (Static content is requested.), this is called “cacheFirst” but if you request a targeted external URL, this is called “networkFirst”.

- CacheFirst - In this function, if the received request has cached before, the cached response is returned to the page. But if not, a new response requested from the network.
- NetworkFirst - In this function, firstly we can try getting an updated response from the network, if this process completed successfully, the new response will be cached and returned. But if this process fails, we check whether the request has been cached before or not. If a cache exists, it is returned to the page, but if not, this is up to you. You can return dummy content or information messages to the page.

to generate flutter_service_worker.js, run flutter build web


```
C:\Users\Dell\Desktop\society_profiles>flutter build web

Warning: In index.html:43: Manual service worker registration deprecated. Use flutter.js service worker bootstrapping
instead. See https://docs.flutter.dev/platform-integration/web/initialization for more details.
Font asset "CupertinoIcons.ttf" was tree-shaken, reducing it from 257628 to 1172 bytes (99.5% reduction). Tree-shaking
can be disabled by providing the --no-tree-shake-icons flag when building your app.
Font asset "MaterialIcons-Regular.otf" was tree-shaken, reducing it from 1645184 to 8944 bytes (99.5% reduction).
Tree-shaking can be disabled by providing the --no-tree-shake-icons flag when building your app.
Compiling lib/main.dart for the Web... 84.7s
✓ Built build\web

C:\Users\Dell\Desktop\society_profiles>
```

fetch

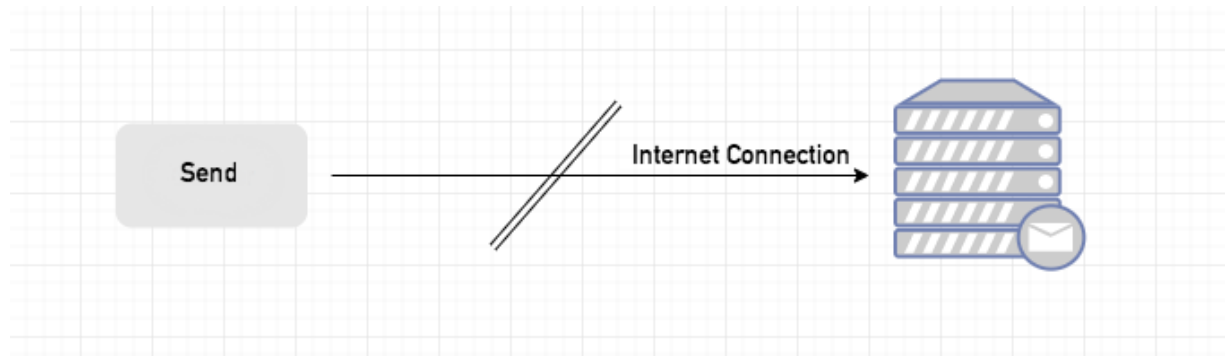
```
self.addEventListener("fetch", (event) => {
  if (event.request.method !== 'GET') {
    return;
  }
  var origin = self.location.origin;
  var key = event.request.url.substring(origin.length + 1);
  // Redirect URLs to the index.html
  if (key.indexOf('?v=') != -1) {
    key = key.split('?v=')[0];
  }
  if (event.request.url == origin || event.request.url.startsWith(origin + '/#') || key == '') {
    key = '/';
  }
  // If the URL is not the RESOURCE list then return to signal that the
  // browser should take over.
  if (!RESOURCES[key]) {
    return;
  }
  // If the URL is the index.html, perform an online-first request.
  if (key == '/') {
    return onlineFirst(event);
  }
  event.respondWith(
    fetch(event.request).then((response) => {
      return caches.open(CACHE_NAME).then((cache) => {
        cache.put(event.request, response.clone());
        return response;
      });
    }).catch(() => {
      return caches.open(CACHE_NAME).then((cache) => cache.match(event.request));
    })
  );
});
```

Sync Event

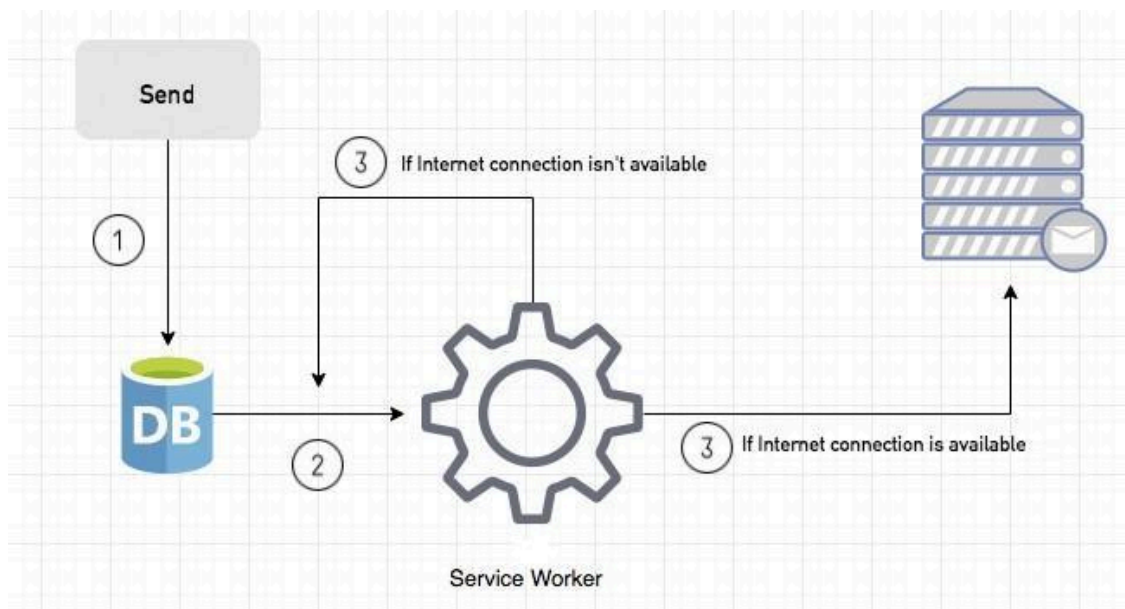
Background Sync is a Web API that is used to delay a process until the Internet connection is stable. We can adapt this definition to the real world; there is an e-mail client application that works on the browser and we want to send an email with this tool. Internet connection is broken while we are writing e-mail content and we didn't realize it. When completing the writing, we click the send button.

Here is a job for the Background Sync.

The following view shows the classical process of sending email to us. If the Internet Connection is broken, we can't send any content to Mail Server.



Here, you can create any scenario for yourself. A sample is in the following for this case.



1. When we click the “send” button, email content will be saved to IndexedDB.
2. Background Sync registration.
3. If the Internet connection is available, all email content will be read and sent to Mail Server.

If the Internet connection is unavailable, the service worker waits until the connection is available even though the window is closed. When it is available, email content will be sent to Mail Server.

You can see the working process within the following code block.

Event Listener for Background Sync Registration
added in main.dart

```
// Register the background sync event using service workers
void registerSync() {
  if (kIsWeb) {
    window.navigator.serviceWorker?.ready.then((registration) {
      registration.sync?.register('sync-data').then((_) {
        print("Sync registered successfully");
      }).catchError((error) {
        print("Sync registration failed: $error");
      });
    }).catchError((error) {
      print("Service worker not ready: $error");
    });
  }
}
```

inside flutter_service_worker.js

```
self.addEventListener('sync', (event) => {
  if (event.tag === 'sync-data') {
    event.waitUntil(syncPendingData());
  }
});

async function syncPendingData() {
  await fetch('/api/sync', { method: 'POST' });
}
```

Push Event

This is the event that handles push notifications that are received from the server. You can apply any method with received data.

We can check in the following example.

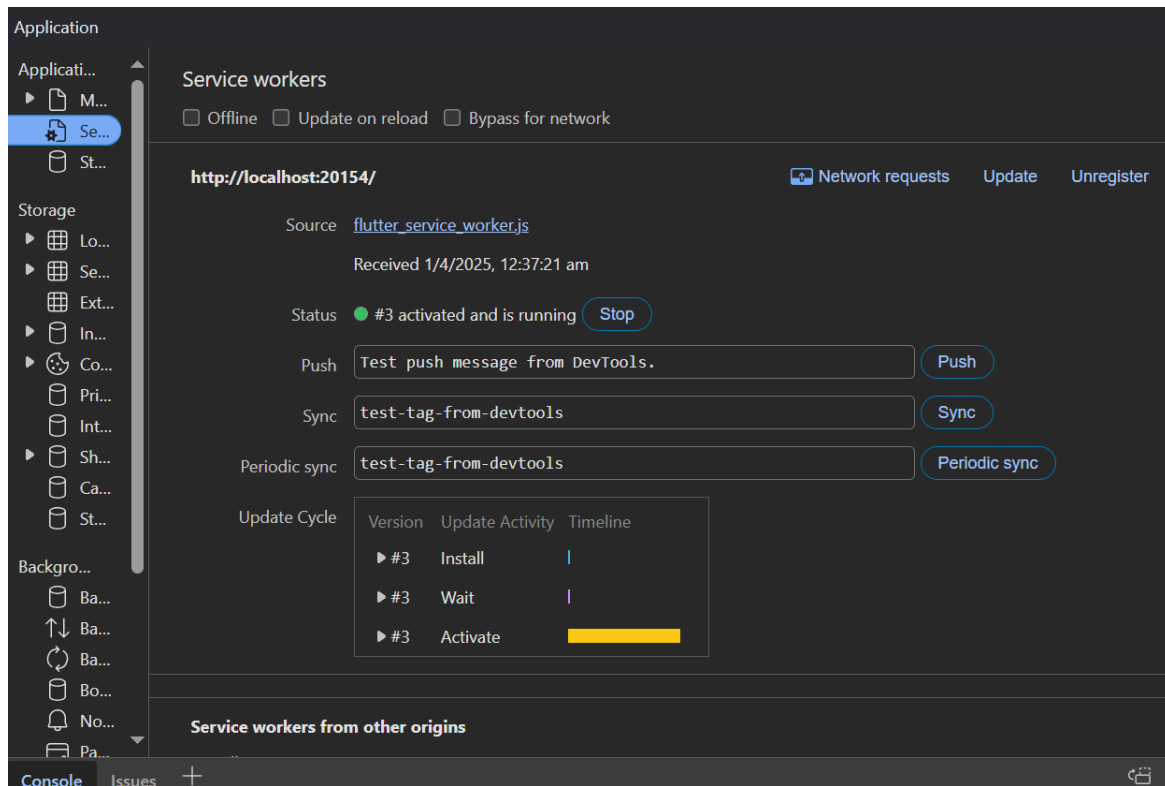
“Notification.requestPermission();” is the necessary line to show notification to the user. If you don't want to show any notification, you don't need this line.

In the following code block is in sw.js file. You can handle push notifications with this event. In this example, I kept it simple. We send an object that has “method” and “message” properties. If the method value is “pushMessage”, we open the information notification with the “message” property.

```
self.addListener('push', (event) => {  
  const options = {  
    body: 'New update in your society!',  
    icon: '/icons/icon-192x192.png',  
    badge: '/icons/icon-72x72.png'  
  };  
  event.waitUntil(  
    self.registration.showNotification('Society App', options)  
  );  
});
```

```
Future<void> subscribeToPushNotifications() async {  
  FirebaseMessaging messaging = FirebaseMessaging.instance;  
  
  // Get the FCM token, used for push notifications  
  String? token = await messaging.getToken();  
  if (token != null) {  
    print('FCM Token: $token'); // Token for push notifications  
  }  
  
  // Handling background messages  
  FirebaseMessaging.onBackgroundMessage((message) async {  
    print('Received background message: $message');  
    // Handle background notification here  
  });  
  
  // Handling foreground messages  
  FirebaseMessaging.onMessage.listen((message) {  
    print('Received foreground message: $message');  
    // Handle foreground notification here  
  });  
}
```

You can use Application Tab from Chrome Developer Tools for testing push notification.

**Conclusion:**

Implementing these events boosted our PWA's responsiveness, reliability, and user communication via push alerts.

Experiment 10**Aim: To study and implement deployment of Ecommerce PWA to GitHub Pages.****Theory:**

GitHub Pages hosts static websites directly from a repository. Flutter web builds can be deployed here using flutter build web and pushing to gh-pages branch.

Deploying E-commerce PWAs with GitHub Pages

GitHub Pages is a free service for hosting static websites directly from a GitHub repository. It's perfect for deploying PWAs.

1. Deployment Overview

- Push the project to GitHub.
- Build static files (npm run build).
- Serve from gh-pages branch.

2. Key Steps













- Setup repository.
- Build the project.
- Use gh-pages or GitHub Actions for deployment.
- Set "homepage" in package.json.

3. Benefits

- Free hosting with CDN.
- No server setup needed.
- Great for front-end and PWA projects.

4. For E-commerce PWAs

Makes the app installable, accessible, and testable in real environments.

 android	Fresh start	2 weeks ago
 assets/flags	Fresh start	2 weeks ago
 ios	Fresh start	2 weeks ago
 lib	more progress	2 weeks ago
 linux	Fresh start	2 weeks ago
 macos	Fresh start	2 weeks ago
 test	Fresh start	2 weeks ago
 web	Fresh start	2 weeks ago
 windows	Fresh start	2 weeks ago
 .env	Fresh start	2 weeks ago
 .gitignore	Fresh start	2 weeks ago
 .metadata	Fresh start	2 weeks ago

Conclusion:

Deployment to GitHub Pages provided a free, reliable hosting solution for showcasing the Society App online.

Experiment 11

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

Theory:

Google Lighthouse audits PWAs for performance, accessibility, SEO, and PWA compliance. It provides improvement tips through scores and diagnostics.

Using Google Lighthouse for PWA Testing

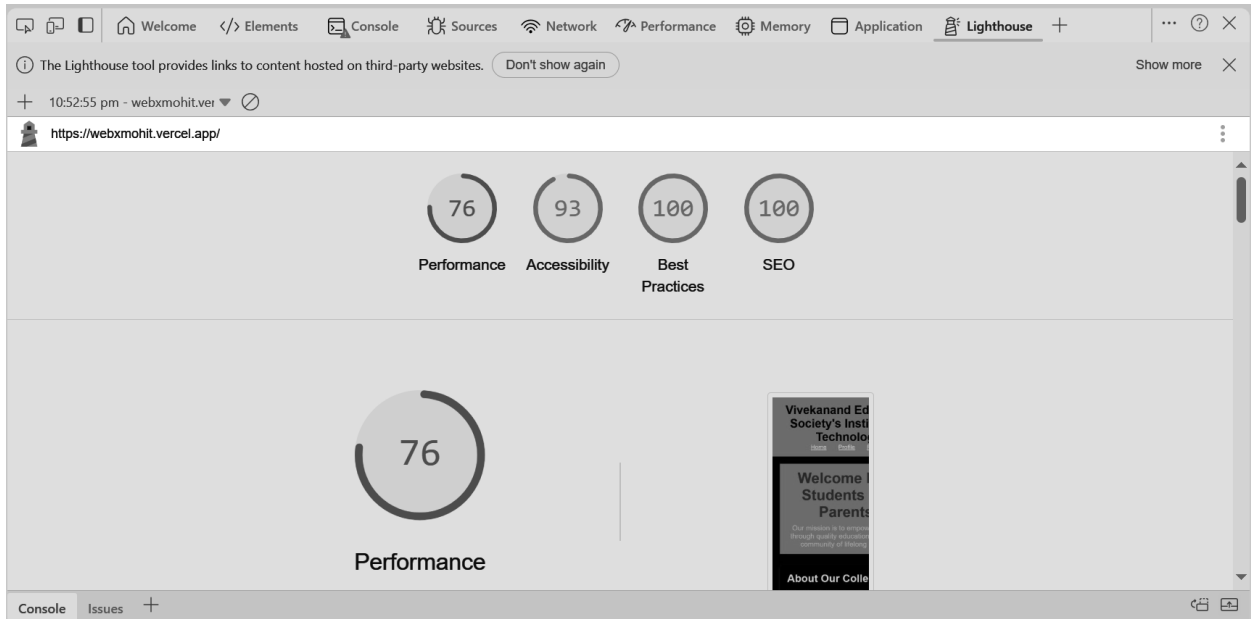
Google Lighthouse is a free tool built into Chrome DevTools that audits web apps for performance, accessibility, SEO, and PWA compliance.

Steps to Test PWA Functionality:

1. Open Chrome DevTools
Right-click on the page → Inspect → Go to the "Lighthouse" tab.
2. Select Categories
Choose "Progressive Web App" along with other desired categories.
3. Run Audit
Click "Analyze page load". Lighthouse will generate a report.
4. Review Results
Check PWA score and key metrics like installability, service worker usage, and offline support.

Benefits:

- Identifies missing PWA features.
- Offers improvement suggestions.
- Helps validate PWA readiness for deployment.

**Conclusion:**

Lighthouse analysis helped ensure that our Society App met core PWA criteria and improved overall performance metrics.