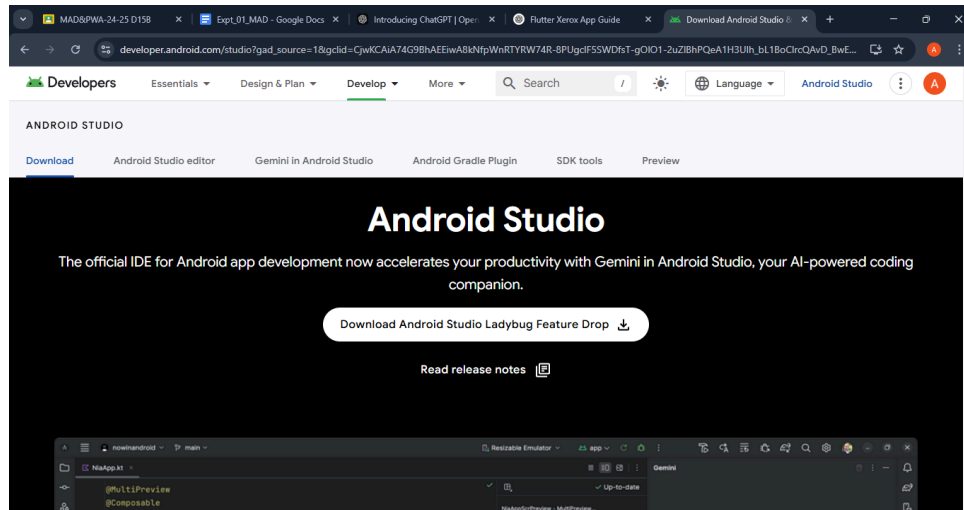


# MAD & PWA

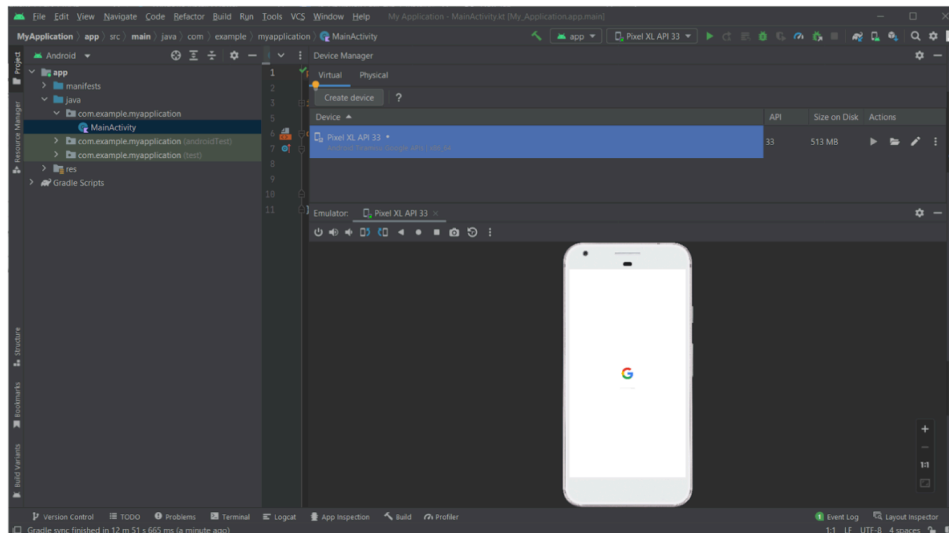
## Experiment No: 01

**Aim: To setup Android App Development Environment.**

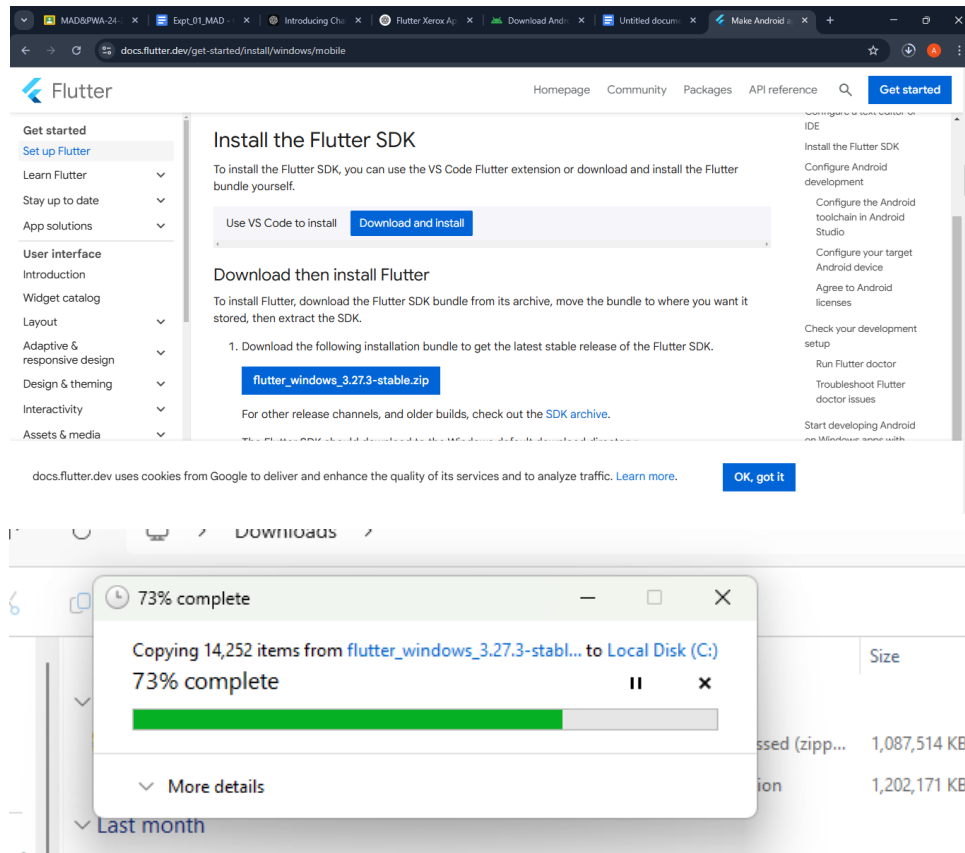
**Step 1: Download Android Studio, run exe file and follow on screen steps to install it.**



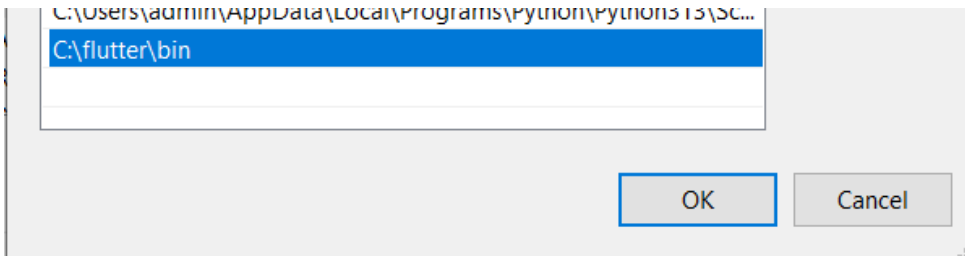
**Step 2: set up an Android emulator in Setting>Virtual Device Manager.**



**Step 3: Download and install Flutter SDK by Extracting zip file(preferably in location C:\flutter).**



**Step 4:** Set path of flutter sdk bin in System path to access flutter in command prompt.



**Step 5:** Run flutter and flutter doctor command to ensure everything is setup for development.

```
C:\Users\admin>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 3.27.4, on Microsoft Windows [Version 10.0.19045.5371], locale en-IN)
[✓] Windows Version (Installed version of Windows is version 10 or higher)
[✓] Android toolchain - develop for Android devices (Android SDK version 35.0.1)
[✓] Chrome - develop for the web
[X] Visual Studio - develop Windows apps
    X Visual Studio not installed; this is necessary to develop Windows apps.
      Download at https://visualstudio.microsoft.com/downloads/.
      Please install the "Desktop development with C++" workload, including all of its default components
[✓] Android Studio (version 2024.2)
[✓] VS Code (version 1.97.0)
[✓] Connected device (3 available)
[✓] Network resources

! Doctor found issues in 1 category.

C:\Users\admin>
```

# MPL Lab 2

Mayur Jaiswal

D15B

24

**Aim:** To design flutter UI using common widgets

**Theory:** Flutter is a UI toolkit by Google used to build natively compiled applications using a single codebase. Flutter provides a rich set of **predefined widgets** that follow a declarative approach to UI design. Widgets like Text, Container, Row, Column, Image, and ListView are the building blocks of any UI in Flutter. These widgets can be combined, styled, and customized to build responsive and attractive user interfaces.

Flutter apps are structured as a **widget tree**, where each UI component is a widget (even layouts and styling). Using these widgets, developers can design both simple and complex UIs efficiently.

---

## Steps to Design Flutter UI Using Common Widgets

1. **Set Up Flutter Project:**
2. **Use MaterialApp as Root Widget:**
3. **Design UI with Common Widgets in HomePage:**
4. **Use Layout Widgets for Positioning:**
  - Column and Row for vertical/horizontal layout.
  - Padding, SizedBox, and Align for spacing and alignment.
5. **Style UI Elements:**
  - Use TextStyle, BoxDecoration, and ButtonStyle.
  - Customize using ThemeData in MaterialApp.

## Code :

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

class LoginScreen extends StatefulWidget {
  @override
  _LoginScreenState createState() => _LoginScreenState();
}

class _LoginScreenState extends State<LoginScreen> {
  final TextEditingController emailController = TextEditingController();
  final TextEditingController passwordController = TextEditingController();
  bool isLoading = false;
```

```

void loginUser(BuildContext context) async {
  setState(() => isLoading = true);
  try {
    await Provider.of<AuthProvider>(context, listen: false)
      .login(emailController.text, passwordController.text);
    Navigator.pushReplacementNamed(context, '/home');
  } catch (e) {
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(
        content: Text("Login Failed: ${e.toString()}"),
        backgroundColor: Colors.redAccent,
      ),
    );
  }
  setState(() => isLoading = false);
}

```

```

@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Container(
      width: double.infinity,
      decoration: BoxDecoration(
        gradient: LinearGradient(
          begin: Alignment.topCenter,
          end: Alignment.bottomCenter,
          colors: [Colors.blueAccent, Colors.lightBlue.shade200],
        ),
      ),
    child: Center(
      child: Padding(
        padding: EdgeInsets.all(20.0),
        child: Card(
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.circular(20),
          ),
          elevation: 8,
          color: Colors.white,
          child: Padding(
            padding: EdgeInsets.all(25.0),
            child: Column(
              mainAxisAlignment: MainAxisAlignment.min,
              children: [
                SizedBox(height: 20),
                Text(
                  "Welcome Back!",
                  style: TextStyle(

```

```

        fontSize: 24,
        fontWeight: FontWeight.bold,
        color: Colors.blueAccent,
      ),
    ),
    SizedBox(height: 20),
    _buildTextField(
      controller: emailController,
      label: "Email"),
    SizedBox(height: 15),
    _buildTextField(
      controller: passwordController,
      label: "Password",
      isPassword: true),
    SizedBox(height: 20),
    isLoading
      ? CircularProgressIndicator()
      : ElevatedButton(
          onPressed: () => loginUser(context),
          child: Text("Login"),
          style: ElevatedButton.styleFrom(
            padding: EdgeInsets.symmetric(vertical: 12, horizontal: 60),
            backgroundColor: Colors.transparent, // Make background transparent
            side: BorderSide(color: Colors.blueAccent), // Optional: Adds a border to the button
          ),
        ),
    SizedBox(height: 15),
    TextButton(
      onPressed: () {}, // Add forgot password logic
      child: Text(
        "Forgot Password?",
        style: TextStyle(color: Colors.blueAccent),
      ),
    ),
  ),
],
),
),
),
),
),
),
),
);
}

```

```

Widget _buildTextField(
  {required TextEditingController controller, required String label, bool isPassword = false}) {
  return TextField(
    controller: controller,

```

```
obscureText: isPassword,  
decoration: InputDecoration(  
  labelText: label,  
  filled: true,  
  fillColor: Colors.white,  
  contentPadding: EdgeInsets.symmetric(horizontal: 20, vertical: 14),  
  border: OutlineInputBorder(  
    borderRadius: BorderRadius.circular(12),  
    borderSide: BorderSide(color: Colors.blueAccent),  
  ),  
  enabledBorder: OutlineInputBorder(  
    borderRadius: BorderRadius.circular(12),  
    borderSide: BorderSide(color: Colors.blueAccent.shade100),  
  ),  
  focusedBorder: OutlineInputBorder(  
    borderRadius: BorderRadius.circular(12),  
    borderSide: BorderSide(color: Colors.blueAccent, width: 2),  
  ),  
),  
);  
}  
}
```

Output:

localhost:8025/#/login

Welcome Back!

Email

Password

Login

Forgot Password?

# MPL Lab 3

Mayur Jaiswal

D15B

24

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

**Theory:** Flutter allows developers to customize their apps by adding **icons**, **images**, and **custom fonts**. These visual elements enhance the user interface and make the app more appealing and intuitive.

- **Icons:** Flutter provides built-in icons via the Icons class. Custom icon packs (like FontAwesome) can also be added.
- **Images:** Images can be local (from assets) or loaded from the internet.
- **Fonts:** You can use Google Fonts or any custom TTF font by declaring them in pubspec.yaml.

---

## Steps:

### 1. Add Icons:

---

### 2. Add Local Images:

**Step A:** Create an assets/images/ folder and add image files (e.g. logo.png).

**Step B:** Declare in pubspec.yaml:

**Step C:** Use in app:

---

### 3. Add Custom Fonts:

**Step A:** Create assets/fonts/ and add .ttf files.

**Step B:** Declare in pubspec.yaml:

**Step C:** Use in Text widget:

## Code :

### Login Page:

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

```
class LoginScreen extends StatefulWidget {
  @override
```



```

_LoginScreenState createState() => _LoginScreenState();
}

```

```

class _LoginScreenState extends State<LoginScreen> {
  final TextEditingController emailController = TextEditingController();
  final TextEditingController passwordController = TextEditingController();
  bool isLoading = false;

```

```

  void loginUser(BuildContext context) async {
    setState(() => isLoading = true);
    try {
      await Provider.of<AuthProvider>(context, listen: false)
        .login(emailController.text, passwordController.text);
      Navigator.pushReplacementNamed(context, '/home');
    } catch (e) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text("Login Failed: ${e.toString()}"),
          backgroundColor: Colors.redAccent,
        ),
      );
    }
    setState(() => isLoading = false);
  }

```

```

@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Container(
      width: double.infinity,
      decoration: BoxDecoration(
        gradient: LinearGradient(
          begin: Alignment.topCenter,
          end: Alignment.bottomCenter,
          colors: [Colors.blueAccent, Colors.lightBlue.shade200],
        ),
      ),
    child: Center(
      child: Padding(
        padding: EdgeInsets.all(20.0),
        child: Card(
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.circular(20),
          ),
          elevation: 8,
          color: Colors.white,
          child: Padding(
            padding: EdgeInsets.all(25.0),

```

```

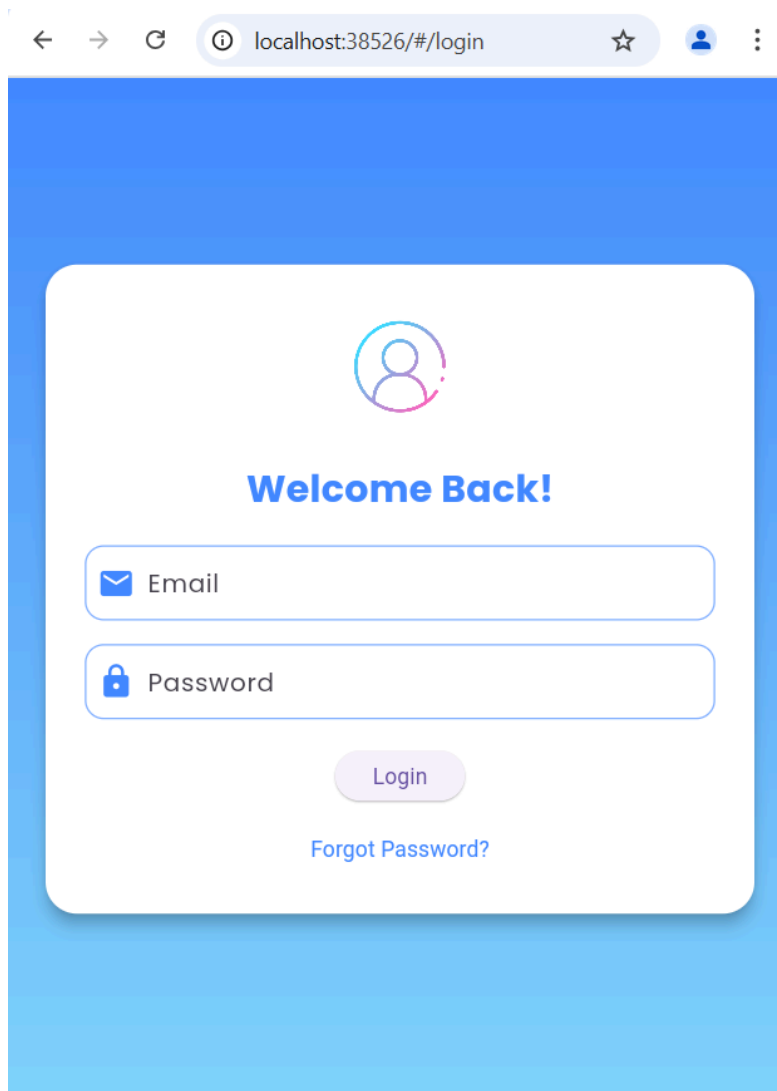
child: Column(
  mainAxisAlignment: MainAxisAlignment.min,
  children: [
    // Use a web image (URL)
    Image.network('https://cdn-icons-png.flaticon.com/512/5087/5087579.png', height: 80),
    SizedBox(height: 20),
    Text(
      "Welcome Back!",
      style: GoogleFonts.poppins(
        fontSize: 24,
        fontWeight: FontWeight.bold,
        color: Colors.blueAccent,
      ),
    ),
    SizedBox(height: 20),
    _buildTextField(
      controller: emailController,
      label: "Email",
      icon: Icons.email,
    ),
    SizedBox(height: 15),
    _buildTextField(
      controller: passwordController,
      label: "Password",
      isPassword: true,
      icon: Icons.lock,
    ),
    SizedBox(height: 20),
    isLoading
      ? CircularProgressIndicator()
      : CustomButton(
          text: "Login",
          onPressed: () => loginUser(context),
        ),
    SizedBox(height: 15),
    TextButton(
      onPressed: () {}, // Add forgot password logic
      child: Text(
        "Forgot Password?",
        style: TextStyle(color: Colors.blueAccent),
      ),
    ),
  ],
),
),
),
),
),
),
),
),
),
);

```


```
}
```

```
Widget _buildTextField(  
  {required TextEditingController controller,  
  required String label,  
  bool isPassword = false,  
  IconData? icon}) {  
  return TextField(  
    controller: controller,  
    obscureText: isPassword,  
    decoration: InputDecoration(  
      prefixIcon: icon != null ? Icon(icon, color: Colors.blueAccent) : null,  
      labelText: label,  
      labelStyle: GoogleFonts.poppins(fontSize: 16),  
      filled: true,  
      fillColor: Colors.white,  
      contentPadding: EdgeInsets.symmetric(horizontal: 20, vertical: 14),  
      border: OutlineInputBorder(  
        borderRadius: BorderRadius.circular(12),  
        borderSide: BorderSide(color: Colors.blueAccent),  
      ),  
      enabledBorder: OutlineInputBorder(  
        borderRadius: BorderRadius.circular(12),  
        borderSide: BorderSide(color: Colors.blueAccent.shade100),  
      ),  
      focusedBorder: OutlineInputBorder(  
        borderRadius: BorderRadius.circular(12),  
        borderSide: BorderSide(color: Colors.blueAccent, width: 2),  
      ),  
    ),  
  );  
}
```


**Output:**




← → ↻ ⓘ localhost:38526/#/login ☆ 👤 ⋮



**Welcome Back!**

 Email

 Password

Login

[Forgot Password?](#)

**MPL Lab 4**  
**Mayur Jaiswal**  
**D15B**  
**24**

**Aim:** Develop an interactive form in Flutter utilizing the Form and TextFormField widgets to efficiently handle user input and validation.

**Theory:** In Flutter, forms are essential for collecting and validating user input. The Form widget acts as a container for form fields, managing their state and validation. TextFormField is a commonly used widget for single-line text input, offering built-in validation and state management. By combining these widgets, developers can create robust forms that provide real-time feedback to users, enhancing the overall user experience.

**Steps to Perform:**

**1. Set Up a New Flutter Project:**

- Create a new Flutter application using the command: `flutter create form_app`.
- Navigate to the project directory: `cd form_app`.

**2. Design the Form Interface:**

In `lib/main.dart`, import the necessary packages:

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

○

Define the main function and set the home to `MyFormPage`:

```
void main() {  
  runApp(MaterialApp(  
    home: MyFormPage(),  
  ));  
}
```

○

- Create a stateful widget `MyFormPage` that returns a `Scaffold` with an `AppBar` and a `Form` widget.

**3. Implement Form Fields with Validation:**

- Within the `Form` widget, add `TextFormField` widgets for each input field (e.g., name, email, password).
- Assign a `TextEditingController` to each field to manage the input.

Add validation logic using the `validator` property:

```
TextFormField(  
  controller: _emailController,  
  decoration: InputDecoration(labelText: 'Email'),  
  validator: (value) {
```

```

    if (value == null || value.isEmpty) {
        return 'Please enter your email';
    }
    if (!RegExp(r'^[@]+\.[^@]+\.[^@]+').hasMatch(value)) {
        return 'Enter a valid email';
    }
    return null;
  },
),

```

○

#### 4. Add a Submit Button:

Include an ElevatedButton that, when pressed, triggers the form's validation and processes the input if valid:

```

ElevatedButton(
  onPressed: () {
    if (_formKey.currentState!.validate()) {
      // Process data
    }
  },
  child: Text('Submit'),
),

```

○

#### 5. Manage Form State:

Use a GlobalKey<FormState> to manage the form's state and validation:

```
final _formKey = GlobalKey<FormState>();
```

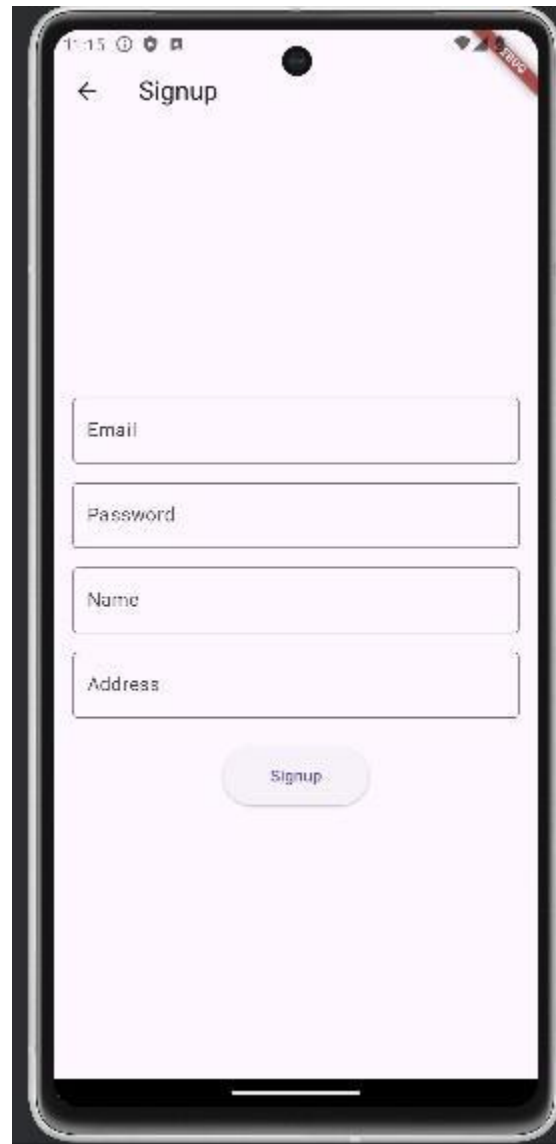
○

- Wrap the Form widget with a Form widget and assign the \_formKey to it.

#### Key Features:

- **State Management:** Utilizes GlobalKey to manage form state effectively.
- **Validation:** Provides real-time validation feedback to users.
- **User Experience:** Enhances user interaction with clear error messages and responsive input fields.

**Output:**



```
class LoginScreen extends StatefulWidget {  
  @override  
  _LoginScreenState createState() => _LoginScreenState();  
}  
  
class _LoginScreenState extends State<LoginScreen> {  
  final _formKey = GlobalKey<FormState>();  
  final _emailController = TextEditingController(); final  
  _passwordController = TextEditingController();  
  
  Future<void> _login() async {  
    if (_formKey.currentState!.validate()) {  
      try {
```

```

        await FirebaseAuth.instance.signInWithEmailAndPassword(
            email: _emailController.text,
            password: _passwordController.text,
        );
        Navigator.pushReplacement(
            context, MaterialPageRoute(builder: (context) => HomeScreen()));
    } on FirebaseAuthException catch (e) {
        if (mounted) {
            ScaffoldMessenger.of(context).showSnackBar(
                SnackBar(content: Text('Failed to login: ${e.message}')),
            );
        }
    }
}

@override
Widget build(BuildContext context) {
    return Scaffold(
        appBar: AppBar(title: Text('Login')),
        body: Padding(
            padding: const EdgeInsets.all(16.0),
            child: Form(
                key: _formKey,
                child: Column(
                    mainAxisAlignment: MainAxisAlignment.center,
                    children: [
                        TextFormField(
                            controller: _emailController,
                            decoration: InputDecoration(
                                labelText: 'Email',
                                border: OutlineInputBorder(),
                            ),
                            validator: (value) => value!.isEmpty ? 'Enter email' : null,
                        ),
                        SizedBox(height: 16),
                        TextFormField(
                            controller: _passwordController,
                            obscureText: true,
                            decoration: InputDecoration(
                                labelText: 'Password',
                                border: OutlineInputBorder(),
                            ),
                        ),
                    ],
                ),
            ),
        ),
    );
}

```



**MPL Lab 5**  
**Mayur Jaiswal**  
**D15B**  
**24**

**Aim:** Implement navigation between multiple screens and incorporate gesture detection to enhance user interaction in a Flutter application.

**Theory:** Navigation and routing are fundamental for multi-screen applications. Flutter's **Navigator** widget manages a stack of routes, enabling seamless transitions between screens. Named routes provide a cleaner way to define and navigate to different screens. Additionally, gestures like taps, swipes, and pinches are integral to modern app interfaces. Flutter's GestureDetector widget allows developers to capture and respond to various user gestures, making the app more interactive and responsive.

**Steps to Perform:**

**1. Set Up a New Flutter Project:**

- Create a new Flutter application: flutter create navigation\_app.
- Navigate to the project directory: cd navigation\_app.[Flutter documentation+2Firebase+2Firebase+2](#)

**2. Define Named Routes:**

In lib/main.dart, set up the MaterialApp with named routes:

```
void main() {  
  runApp(MaterialApp(  
    initialRoute: '/',  
    routes: {  
      '/': (context) => HomeScreen(),  
      '/second': (context) => SecondScreen(),  
    },  
  ));  
}
```

**3. Create Screens:**

- Define two stateless widgets, HomeScreen and SecondScreen, each returning a Scaffold with an AppBar and body content.

**4. Implement Navigation:**

In HomeScreen, add a button that navigates to SecondScreen when pressed:

```
ElevatedButton(  
  onPressed: () {  
    Navigator.pushNamed(context, '/second');  
  },  
  child: Text('Go to Second Screen'),  
),
```

○

In SecondScreen, add a button to navigate back:

```
ElevatedButton(
  onPressed: () {
    Navigator.pop(context);
  },
  child: Text('Go Back'),
),
```

## 5. Handle Gestures:

Wrap widgets with GestureDetector to detect gestures:

```
GestureDetector(
  onTap: () {
    // Handle tap
  },
  child: Container(
    color: Colors.blue,
    height: 100,
    width: 100,
  ),
),
```

---

## 2. Applying Navigation, Routing, and Gestures in a Flutter App (*continued*)

### 5. Handle Gestures (continued):

You can use GestureDetector to capture various gesture events such as onTap, onDoubleTap, onLongPress, onPanUpdate, etc. This can be useful for creating interactive UI components

```
GestureDetector(
  onTap: () {
    print('Box tapped!');
  },
  onDoubleTap: () {
    print('Box double tapped!');
  },
  child: Container(
    color: Colors.amber,
    height: 100,
    width: 100,
    child: Center(child: Text("Tap Me")),
  ),
),
```

### 6. Navigate with Parameters (Optional):

```
Navigator.push(
  context,
  MaterialPageRoute(
    builder: (context) => SecondScreen(data: 'Hello'),
  ),
);
```

In the second screen:

```
class SecondScreen extends StatelessWidget {  
  final String data;  
  SecondScreen({required this.data});  
  ...  
}
```

### Key Features:

- **Named & Anonymous Routing:** Manage multiple pages easily.
- **Gesture Detection:** Supports tap, swipe, drag, long press, etc.
- **Flexible Navigation Stack:** Push, pop, replace, and popUntil available.
- **Parameter Passing:** Allows context-aware navigation.

Once Logged in the app gets navigated to the homescreen and “addskill” button navigates to the addskillscreen

```
// screens/login_screen.dart
import 'package:flutter/material.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'home_screen.dart';
import 'signup_screen.dart';

class LoginScreen extends StatefulWidget {
  @override
  _LoginScreenState createState() => _LoginScreenState();
}

class _LoginScreenState extends State<LoginScreen> {
  final _formKey = GlobalKey<FormState>();
  final _emailController = TextEditingController();
  final _passwordController =
    TextEditingController();

  Future<void> _login() async {
    if (_formKey.currentState!.validate()) {
      try {
        await FirebaseAuth.instance.signInWithEmailAndPassword(
          email: _emailController.text,
          password: _passwordController.text,
        );
        Navigator.pushReplacement(
          context, MaterialPageRoute(builder: (context) => HomeScreen()));
      } on FirebaseAuthException catch (e) {
        if (mounted) {
          ScaffoldMessenger.of(context).showSnackBar(
            SnackBar(content: Text('Failed to login: ${e.message}')),
          );
        }
      }
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(title: Text('Login')),
    );
  }
}
```

```

body: Padding(
  padding: const EdgeInsets.all(16.0),
  child: Form(
    key: _formKey,
    child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        TextFormField(
          controller: _emailController,
          decoration: InputDecoration(
            labelText: 'Email',
            border: OutlineInputBorder(),
          ),
          validator: (value) => value!.isEmpty ? 'Enter email' : null,
        ),
        SizedBox(height: 16),
        TextFormField(
          controller: _passwordController,
          obscureText: true,
          decoration: InputDecoration(
            labelText: 'Password',
            border: OutlineInputBorder(),
          ),
          validator: (value) => value!.isEmpty ? 'Enter password' : null,
        ),
        SizedBox(height: 24),
        ElevatedButton(
          onPressed: _login,
          child: Text('Login'),
          style: ElevatedButton.styleFrom(
            padding: EdgeInsets.symmetric(horizontal: 40, vertical: 15),
          ),
        ),
        SizedBox(height: 12),
        TextButton(
          onPressed: () {
            Navigator.push(context,
              MaterialPageRoute(builder: (context) => SignupScreen()));
          },
          child: Text("Signup"))
      ],
    ),
  ),
),

```

```

    );
  }
}
// screens/add_skill_screen.dart
import 'package:flutter/material.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'skill.dart';
import 'database_service.dart';
import 'user_model.dart';

class AddSkillPage extends StatefulWidget {
  final String userName;

  const AddSkillPage({Key? key, required this.userName}) : super(key: key);

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

class _AddSkillPageState extends State<AddSkillPage> {
  final _skillController = TextEditingController();
  final DatabaseService _databaseService = DatabaseService();
  UserModel? user;
  String? address;

  @override
  void initState() {
    super.initState();
    _loadUserData();
  }

  Future<void> _loadUserData() async {
    final currentUser = FirebaseAuth.instance.currentUser;
    if (currentUser != null) {
      try {
        user = await _databaseService.getUser(currentUser.uid);
        setState(() {
          address = user?.address;
        });
      } catch (error) {
        print('Error loading user data: $error');
      }
    }
  }
}

```

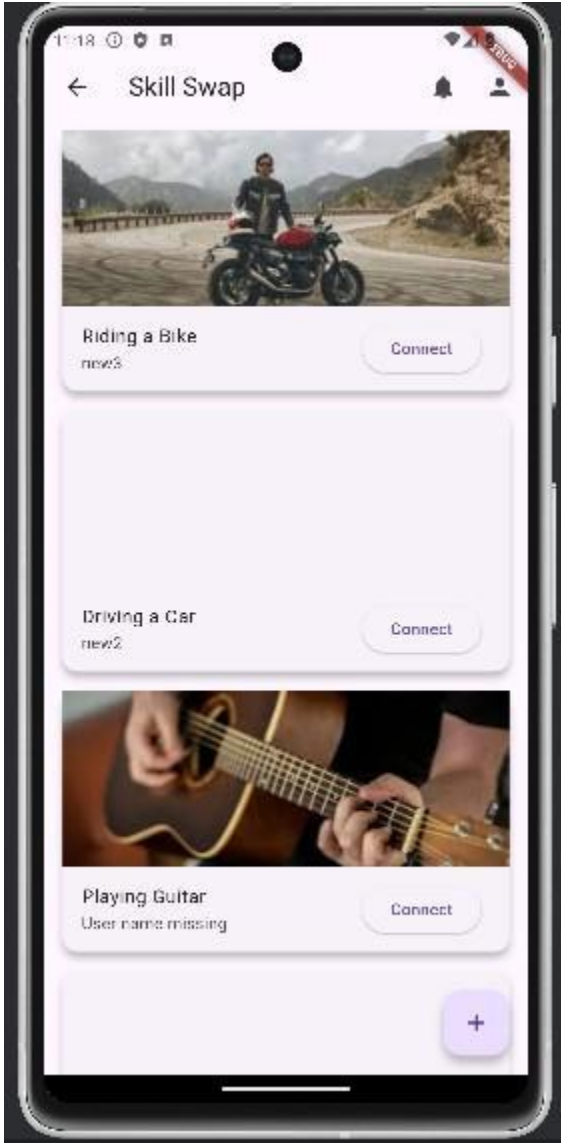
```
}
```

```
Future<void> _addSkill() async {  
  try {  
    final userId = FirebaseAuth.instance.currentUser!.uid;  
    final skill = Skill(  
      userId: userId,  
      userName: widget.userName,  
      skillName: _skillController.text,  
      address: address ?? "No Address"  
    );  
    await _databaseService.addSkill(skill);  
    if (mounted) {  
      Navigator.pop(context);  
    }  
  } catch (e) {  
    if (mounted) {  
      ScaffoldMessenger.of(context).showSnackBar(  
        SnackBar(content: Text('Failed to add skill: ${e.toString()}')),  
      );  
    }  
  }  
}
```

```
@override  
Widget build(BuildContext context) {  
  return Scaffold(  
    appBar: AppBar(title: Text('Add Skill')),  
    body: Padding(  
      padding: const EdgeInsets.all(16.0),  
      child: Column(  
        children: [  
          TextField(  
            controller: _skillController,  
            decoration: InputDecoration(labelText: 'Skill'),  
          ),  
          SizedBox(height: 16),  
          ElevatedButton(onPressed: _addSkill, child: Text('Add')),  
        ],  
      ),  
    ),  
  );  
}
```







**Aim:** To integrate Firebase Firestore with a Flutter application in order to store, retrieve, and manage user data in real time using a cloud-based NoSQL database.

**Theory:** Firebase Firestore is a scalable, cloud-based NoSQL database that allows mobile and web applications to store and sync data in real-time. In Flutter, this is accomplished using the `cloud_firestore` plugin which provides an API to perform Create, Read, Update, and Delete (CRUD) operations. Firestore organizes data in documents and collections, making it ideal for structured and semi-structured data.

Firestore allows multiple users to see live changes without reloading the app, making it an excellent choice for apps that require collaboration, chat features, or dynamic updates. The integration includes Firebase Authentication for user management and Firestore for data storage.

### **Steps to Perform:**

#### **1. Create and Configure Firebase Project:**

- Visit Firebase Console.
- Create a new project, name it, and disable Google Analytics if not needed.
- Click on "Add App" → select Flutter/Android/iOS as per your platform.
- Register your app and download the `google-services.json` or `GoogleService-Info.plist` file.

#### **2. Add Firebase SDK and Plugins:**

Add the Firebase and Firestore dependencies in `pubspec.yaml`:

dependencies:

`firebase_core: ^2.0.0`

`cloud_firestore: ^4.0.0`

- 
- Run `flutter pub get` to install the packages.

#### **3. Initialize Firebase in `main.dart`:**

Import required packages:

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

○

Modify `main()`:

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

○

#### 4. CRUD Operations with Firestore:

##### Create/Insert Data:

```
FirebaseFirestore.instance.collection('users').add({  
  'name': 'Alice',  
  'email': 'alice@example.com',  
});
```

○

##### Read Data (StreamBuilder):

```
StreamBuilder(  
  stream: FirebaseFirestore.instance.collection('users').snapshots(),  
  builder: (context, snapshot) {  
    if (!snapshot.hasData) return CircularProgressIndicator();  
    return ListView(  
      children: snapshot.data!.docs.map((doc) => Text(doc['name'])).toList(),  
    );  
  },  
);
```

○

##### Update Data:

```
FirebaseFirestore.instance.collection('users').doc(docId).update({  
  'email': 'new@example.com',  
});
```

○

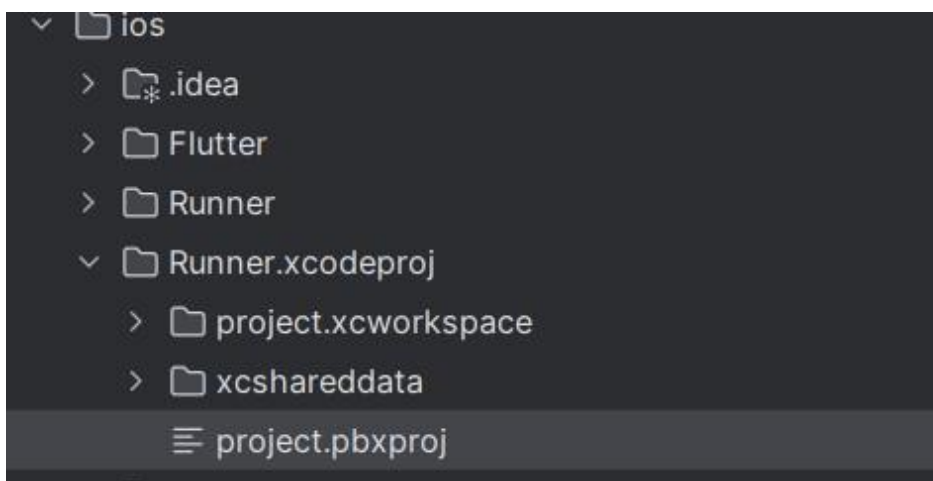
##### Delete Data:

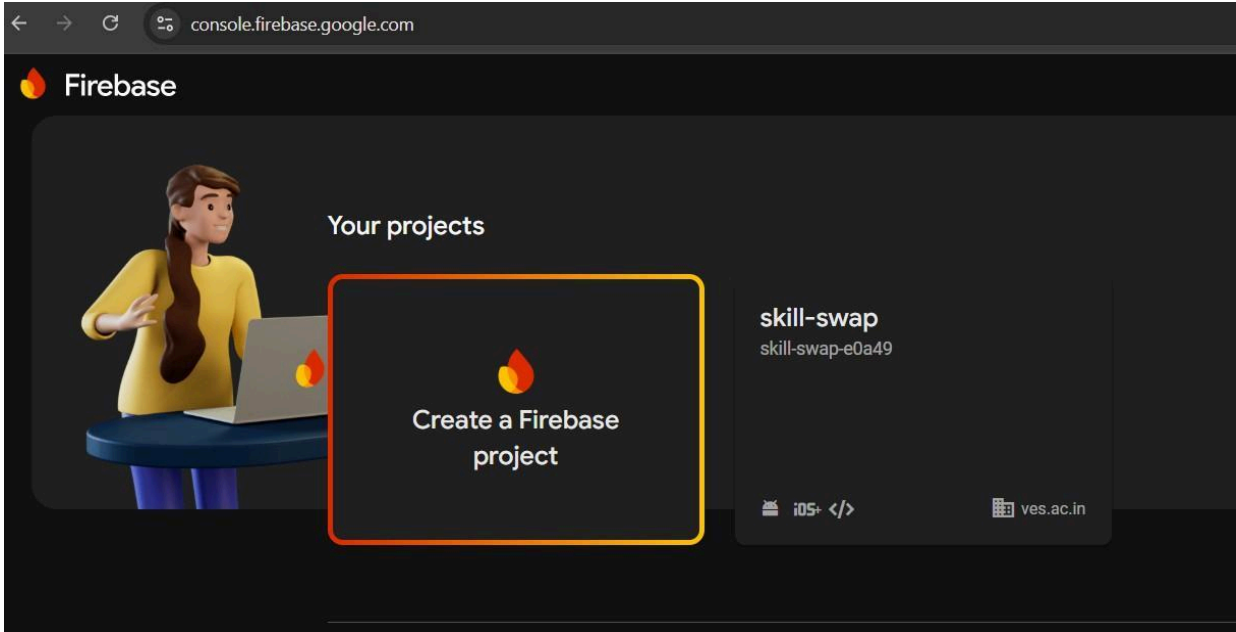
```
FirebaseFirestore.instance.collection('users').doc(docId).delete();
```

○

##### Key Features:

- **Real-Time Syncing:** Data changes instantly reflect across connected clients.
- **Cloud-Hosted and Scalable:** Ideal for apps of all sizes.
- **Simple Integration:** Seamless with Firebase Authentication and Storage.
- **Structured Collections/Documents:** Easy data modeling.
- **Offline Persistence:** Works offline and syncs when reconnected.

















# skill-swap

Spark plan

Getting started

1 app visible (max 3)

4 apps in project

		<b>skill_swap (android)</b> com.example.skill_swap	
		<b>com.example.skillSwap</b>	
		<b>skill_swap (web)</b> Web App	
		<b>skill_swap (windows)</b> Web App	

*Note: all apps are included in project-level metrics below, but only selected apps above are broken out*

**MPL Lab 7**  
**Mayur Jaiswal**  
**D15B**  
**24**

**Aim:** To create and configure a Web App Manifest file that enables Progressive Web App (PWA) features such as “Add to Home Screen” and native-like appearance.

**Theory:** The Web App Manifest is a simple JSON file that provides essential information about a web application (such as name, icons, theme color, and start URL). It allows a browser to install the web application to a device's home screen and launch it in a standalone window without the browser UI, offering a native app-like experience.

The manifest must be linked in the HTML <head> for the browser to recognize and apply it.

**Steps to Perform:**

Create manifest.json: Place this file in the root or public/ directory of your project.

```
json
{
  "name": "My E-Commerce App",
  "short_name": "ShopApp",
  "start_url": "/index.html",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#6200ea",
  "orientation": "portrait",
  "icons": [
    {
      "src": "icons/icon-192x192.png",
      "type": "image/png",
      "sizes": "192x192"
    },
    {
      "src": "icons/icon-512x512.png",
      "type": "image/png",
      "sizes": "512x512"
    }
  ]
}
```

1.

**Link Manifest in HTML: In your index.html:**

```
html
<link rel="manifest" href="manifest.json">
<meta name="theme-color" content="#6200ea">
```

**Add Icons:**

Include icons in the specified sizes inside the /icons folder. These are used for the app shortcut on mobile.

**Test in Browser:**

- Run your app.
- Open Developer Tools → Application tab → Manifest.
- Check for successful registration and “Install” prompt.

### Key Features:

- **App-Like Feel:** Runs in full-screen mode.
- **Custom Branding:** Shows app name and icon on the home screen.
- **Offline Availability:** Combined with service workers, enhances offline UX.
- **Multi-Device Support:** Works on mobile, tablet, desktop.

```
frontend > build > {} manifest.json > ...
1  {
2    "short_name": "MyApp",
3    "name": "My MERN PWA App",
4    "id": "/",
5    "start_url": ".",
6
7    "display": "standalone",
8    "background_color": "#ffffff",
9    "theme_color": "#6200ee",
10   "icons": [
11     {
12       "src": "icons/icon-192x192.png",
13       "sizes": "192x192",
14       "type": "image/png",
15       "purpose": "any"
16     },
17     {
18       "src": "icons/icon-512x512.png",
19       "sizes": "512x512",
20       "type": "image/png",
21       "purpose": "any"
22     }
23   ]
24 }
```

```
PS D:\Downloads\millets-main\millets-main\frontend> serve -s build
```

Serving!

- Local: http://localhost:3000



Application

Manifest

Service workers

Storage

Storage

Local storage

Session storage

Extension storage

IndexedDB

Cookies

Private state tokens

Interest groups

Shared storage

Cache storage

Storage buckets

Background services

Back/forward cache

Background fetch

Background sync

Bounce tracking mitigations

Manifest

App Manifest

manifest.json

Identity

Name

My MERN PWA App

Short name

MyApp

Description

Computed App ID

http://localhost:3000/ [Learn more](#)

Presentation

Start URL

Theme color

#6200ee

Background color

#ffffff

Orientation

Display

standalone

**MPL Lab 8**  
**Mayur Jaiswal**  
**D15B**  
**24**

**Aim:** To create, register, and activate a service worker that provides offline support, cache management, and background capabilities for a Progressive Web App (PWA).

**Theory:** A service worker is a JavaScript file that acts as a proxy between the web app and the network. It intercepts network requests and can serve cached files when offline, greatly improving load performance and reliability. It runs independently of the main web app thread and is capable of managing push notifications, background sync, and cache strategies.

Service workers follow a lifecycle: install, activate, and then fetch. Properly caching assets during install and cleaning up old caches during activation is crucial for version control.

**Steps to Perform:**

Create service-worker.js: In your project root or public folder:

```
javascript

const CACHE_NAME = 'v1';
const urlsToCache = ['/', '/index.html', '/styles.css', '/script.js'];

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

**1.**

**Add Activation Event: Remove old caches when a new service worker is activated.**

```
javascript

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

**2.**

**Register the Service Worker in Your App: In your main JS or HTML:**

```
javascript

if ('serviceWorker' in navigator) {
  navigator.serviceWorker.register('/service-worker.js')
    .then(() => console.log('Service Worker Registered'))
```

```
.catch(err => console.error('Registration failed:', err));  
}
```

**3. Verify Registration: Use Chrome DevTools > Application tab > Service Workers to check status.**

**Key Features:**

- **Offline First: Loads app even with no internet.**
- **Improved Speed: Loads from cache for instant access.**
- **Custom Cache Strategy: You decide which files to cache and when to update.**
- **Persistent Background Control: Works independently from the page.**

← → ↻ 📍 localhost:3000

Millets

Sign in or create an account

Username

Password

Sign In

New user? Register here

🌐

Elements Console Sources Network >>

🔍 Filter

AB Fetch/XHR Doc CSS JS Font Img Media Manifest WS Wasm Other

Name	Status	Type	Initiator	Size	Time
main.c4060b96.css	304	stylesheet	Other	113 B	3 ms
log?format=json&hasfast=true&a...	200	fetch	m:el_main96	151 B	394 ms

2 requests | 264 B transferred | 3.1 kB resources

Console AI assistance What's new X

← → ↻ 📍 localhost:3000

Millets

Sign in or create an account

Username

Password

Sign In

New user? Register here

🌐

Elements Console Sources Network Performance Memory Application Privacy and security Lighthouse >> 1 2 28

🔍 Filter

Application

Manifest Service workers Storage

Storage

Local storage Session storage Extension storage IndexedDB Cookies Private state tokens Interest groups Shared storage

http://localhost:3000

https://securepubads...

https://www.googlea...

https://ep3.adtraffic...

Cache storage

Storage buckets

Background services

Back/forward cache Background fetch Background sync Bounce tracking mi... Notifications Payment handler Periodic background... Speculative loads Push messaging

Shared storage

Origin http://localhost:3000

Creation Time Not yet created

Number of Entries 0

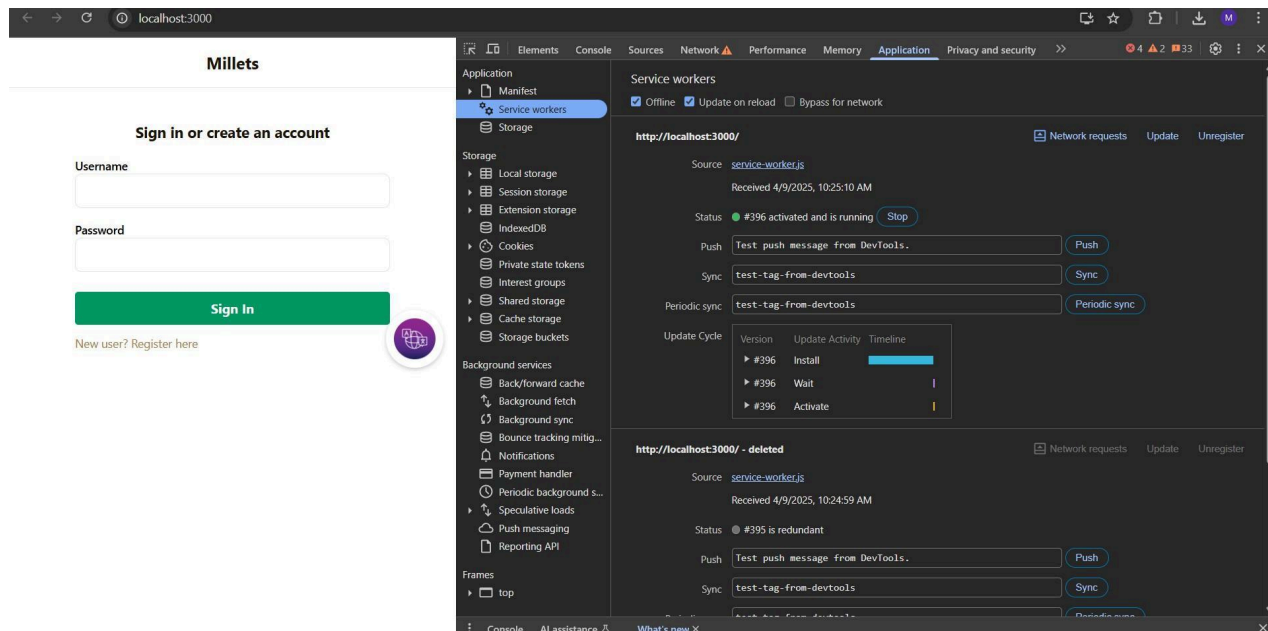
Number of Bytes Used 0

Entropy Budget for Fenced Frames 0

Key	Value
-----	-------

No value selected  
Select a value to preview

Console AI assistance What's new X



## ServiceWorker

- ☐ Open DevTools window and pause JavaScript execution on Service Worker startup for debugging.

**Registrations in: C:\Users\Hp\AppData\Local\Google\Chrome\User Data\Profile 2 (17)**

Scope: chrome-extension://camppjleccjaphfdbohjdohecfnoiekc/

Storage key:

Origin: chrome-extension://camppjleccjaphfdbohjdohecfnoiekc

Top level site: chrome-extension://camppjleccjaphfdbohjdohecfnoiekc

Ancestor chain bit: SameSite

Registration ID: 79

Navigation preload enabled: false

Navigation preload header length: 4

Active worker:

Installation Status: ACTIVATED

Running Status: STOPPED

Fetch handler existence: DOES\_NOT\_EXIST

Fetch handler type: NO\_HANDLER

Script: chrome-extension://camppjleccjaphfdbohjdohecfnoiekc/service-worker-loader.js

Version ID: 362

Renderer process ID: 0

Renderer thread ID: -1

DevTools agent route ID: -2

Log:

Unregister

Start



**MPL Lab 9**  
**Mayur Jaiswal**  
**D15B**  
**24**

**Aim:** To implement advanced service worker events—fetch, sync, and push—for improved performance, background syncing, and notification capabilities in an E-commerce PWA.

**Theory:**

- fetch event: Lets you intercept network requests and serve cached responses or perform custom network strategies (cache-first, network-first).
- sync event: Ensures data is sent to the server when the network is available, useful for retrying failed operations.
- push event: Allows background notifications using the Push API, improving engagement even when the app is not open.

**Steps to Perform:**

Intercept Requests with fetch:

javascript

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

- 1.
2. **Add Background Sync:**

**In service-worker.js:**

javascript

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

```
function sendCartToServer() {  
  // retry pending cart orders  
}
```

○

**In app JS:**

javascript

```
navigator.serviceWorker.ready.then(swReg => {  
  return swReg.sync.register('sync-cart');  
});
```

○

### 3. Push Notifications:

In service-worker.js:

javascript

```
self.addEventListener('push', event => {  
  const data = event.data.json();  
  self.registration.showNotification(data.title, {  
    body: data.body,  
    icon: 'icons/icon-192x192.png',  
  });  
});
```

#### Key Features:

- **Smart Caching:** Improve performance and offline support with fetch control.
- **Background Reliability:** Retry failed tasks when online using sync.
- **User Engagement:** Push notifications improve return rates.
- **Seamless UX:** Users enjoy uninterrupted service even with bad networks.



```
self.addEventListener("fetch", (event) => {
  event.respondWith(
    caches.match(event.request).then((response) => {
      return (
        response ||
        fetch(event.request).catch(() => {
          // fallback page or offline message
          return caches.match("/index.html");
        })
      );
    })
  );
});
```

```
self.addEventListener("sync", (event) => {
  if (event.tag === "sync-post-data") {
    event.waitUntil(sendPendingData());
  }
});

async function sendPendingData() {
  const data = await getPendingDataFromIndexedDB();
  if (data) {
    await fetch("/api/postData", {
      method: "POST",
      body: JSON.stringify(data),
      headers: {
        "Content-Type": "application/json",
      },
    });
    console.log("Pending data sent!");
  }
}
```

Millet's

Sign in or create an account

Username

Password

Sign In

New user? Register here

Application

Manifest

Service workers

Storage

Storage

- Local storage
- Session storage
- Extension storage
- IndexedDB
- Cookies
- Private state tokens
- Interest groups
- Shared storage
- Cache storage
- Storage buckets

Background services

- Back/forward cache
- Background fetch
- Background sync
- Bounce tracking mitig...
- Notifications
- Payment handler
- Periodic background s...
- Speculative loads
- Push messaging
- Reporting API

Frames

- top

Service workers

☒ Offline ☒ Update on reload ☐ Bypass for network

http://localhost:3000/

Network requests Update Unregister

Source [service-worker.js](#)

Received 4/9/2025, 10:25:10 AM

Status ● #396 activated and is running 

Stop

Push 

Test push message from DevTools.

Push

Sync 

test-tag-from-devtools

Sync

Periodic sync 

test-tag-from-devtools

Periodic sync

Update Cycle

Version	Update Activity	Timeline
▶ #396	Install	
▶ #396	Wait	
▶ #396	Activate	

http://localhost:3000/ - deleted

Network requests Update Unregister

Source [service-worker.js](#)

Received 4/9/2025, 10:24:59 AM

Status ● #395 is redundant

Push 

Test push message from DevTools.

Push

Sync 

test-tag-from-devtools

Sync

Periodic sync 

test-tag-from-devtools

Periodic sync

Console

AI assistance

What's new X

**MPL Lab 10**  
**Mayur Jaiswal**  
**D15B**  
**24**

**Aim:** To deploy a production-ready E-commerce PWA to GitHub Pages for public access and testing.

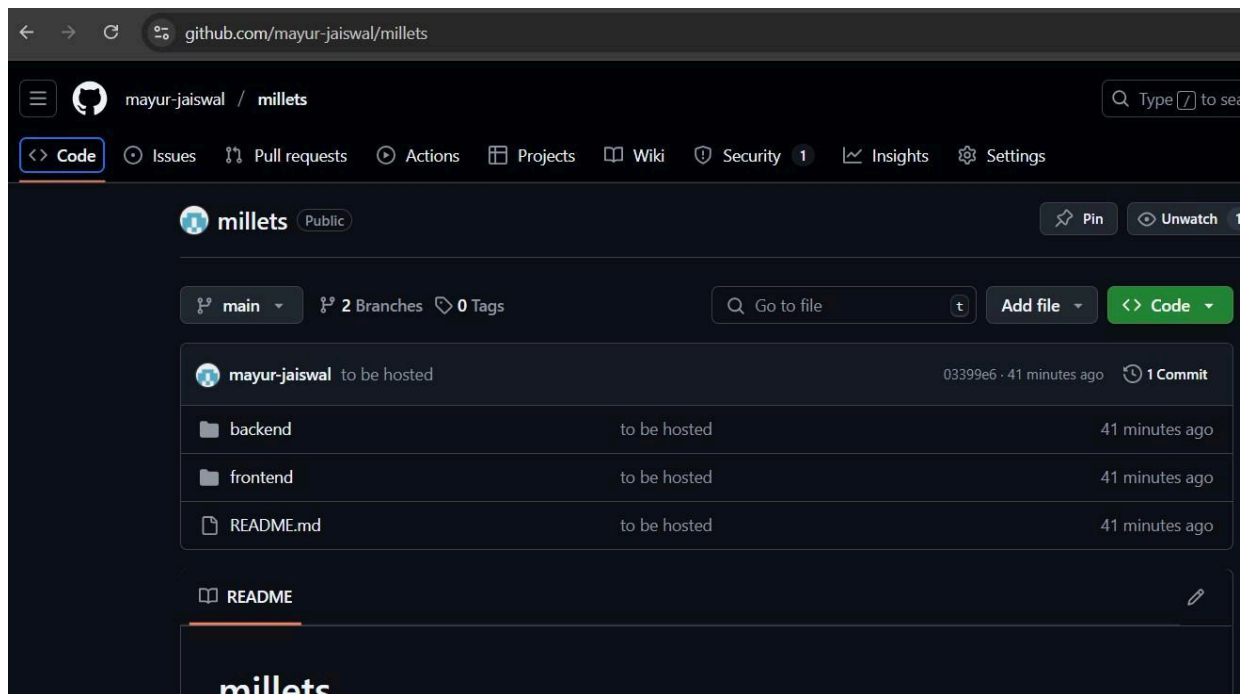
**Theory:** GitHub Pages is a free hosting service for static websites, making it an ideal platform for deploying PWAs. Apps must be built into static files (HTML, CSS, JS) and pushed to the **gh-pages** branch of a GitHub repository or published via tools like **gh-pages** npm package.

**Key Features:**

- No Hosting Fees: Ideal for students, freelancers, and MVPs.
- Instant Preview: Easily share your app with others.
- Custom Domains: Add your own domain via settings.
- Easy Updates: Re-deploy using a single command.

## Steps to Deploy an E-commerce PWA to GitHub Pages

1. Create a GitHub Repository – Set up a new public repository on GitHub.
2. Initialize Git – Link your local project to the repository and commit your code.
3. Push Code to GitHub – Upload your project files to GitHub.
4. Enable GitHub Pages – Go to the repository's Settings > Pages, set the source to the main branch, and save.
5. Install Deployment Tool – Add the necessary package for GitHub Pages deployment.
6. Update Project Settings – Modify the project configuration to specify the deployment URL.
7. Deploy the PWA – Build the project and publish it to GitHub Pages.
8. Access Live Site – Visit the provided GitHub Pages link to view your deployed PWA



millets

Public

Pin

Watch

Fork 0

General

Access

Collaborators

Moderation options

Code and automation

Branches


Tags

Rules

## GitHub Pages

GitHub Pages is designed to host your personal, organization, or project pages from a GitHub repository.

Your site is live at <https://mayur-jaiswal.github.io/millets/>

Last deployed by  mayur-jaiswal 16 minutes ago

Visit site

## Build and deployment

Source

Deploy from a branch

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

**Theory:** Google Lighthouse is an open-source tool built into Chrome DevTools. It provides detailed reports on performance, accessibility, best practices, SEO, and PWA compliance. It simulates loading conditions and gives scores, along with actionable recommendations.

It is crucial for optimizing the user experience, especially for mobile users and slow network conditions.

**Steps to Perform:**

1. Open Your App in Chrome: Ensure your app is running on HTTPS or localhost.
2. Access Lighthouse:
  - Right-click → Inspect → DevTools.
  - Navigate to the Lighthouse tab.
  - Choose the categories: Performance, PWA, Accessibility, etc.
  - Select device type: Mobile/Desktop.
3. Generate Report:
  - Click “Generate Report”.
  - Review your scores and suggestions.
4. Fix Issues:
  - Implement suggestions like image compression, lazy loading, better caching, improved service worker registration, manifest improvements, etc.
5. Re-test After Changes:
  - Re-run Lighthouse to track improvements.

**Key Features:**

- Progressive Web App Audit: Verifies installability, service worker, and offline readiness.
- Performance Optimization: Identifies loading bottlenecks.
- Accessibility Check: Ensures app is usable for everyone.
- Best Practices and SEO: Improves overall quality and searchability.

Generate a Lighthouse report

Analyze page load

Mode [Learn more](#)

- ☒ Navigation (Default)
- ☐ Timespan
- ☐ Snapshot

Device

- ☐ Mobile
- ☒ Desktop

Categories

- ☒ Performance
- ☒ Accessibility
- ☒ Best practices
- ☒ SEO

10:26:51 AM - localhost:3000

http://localhost:3000/

Performance 92 Accessibility 92 Best Practices 56 SEO 100

92

Performance

Values are estimated and may vary. The [performance score](#) is calculated directly from these metrics. [See calculator.](#)

▲ 0-49 ■ 50-89 ● 90-100

Mobile

Sign in or create an account

Username

Password

Sign in

Remember me

## METRICS

[Expand view](#)

■ First Contentful Paint

1.2 s

■ Largest Contentful Paint

1.5 s

● Total Blocking Time

0 ms

● Cumulative Layout Shift

0

■ Speed Index

1.5 s

## DIAGNOSTICS

▲ Eliminate render-blocking resources — Potential savings of 780 ms



▲ Largest Contentful Paint element — 1,450 ms



■ Serve images in next-gen formats — Potential savings of 20 KiB



■ Serve static assets with an efficient cache policy — 3 resources found



■ Properly size images — Potential savings of 33 KiB



■ Reduce unused CSS — Potential savings of 60 KiB



■ Reduce unused JavaScript — Potential savings of 2,978 KiB



○ User Timing marks and measures — 3 user timings



○ Avoid non-composited animations — 1 animated element found



○ Avoid chaining critical requests — 4 chains found



○ Minimize third-party usage — Third-party code blocked the main thread for 10 ms

