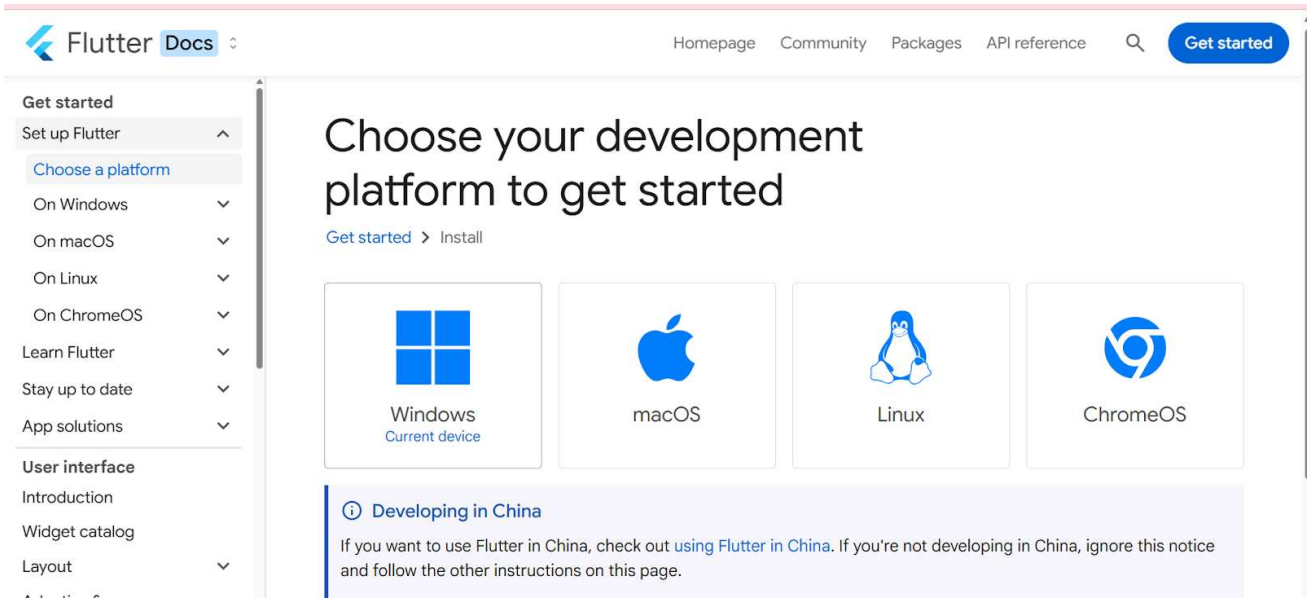
**EXP 1**

**AIM: Installation and Configuration of Flutter Environment.**

**Install the Flutter SDK**

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

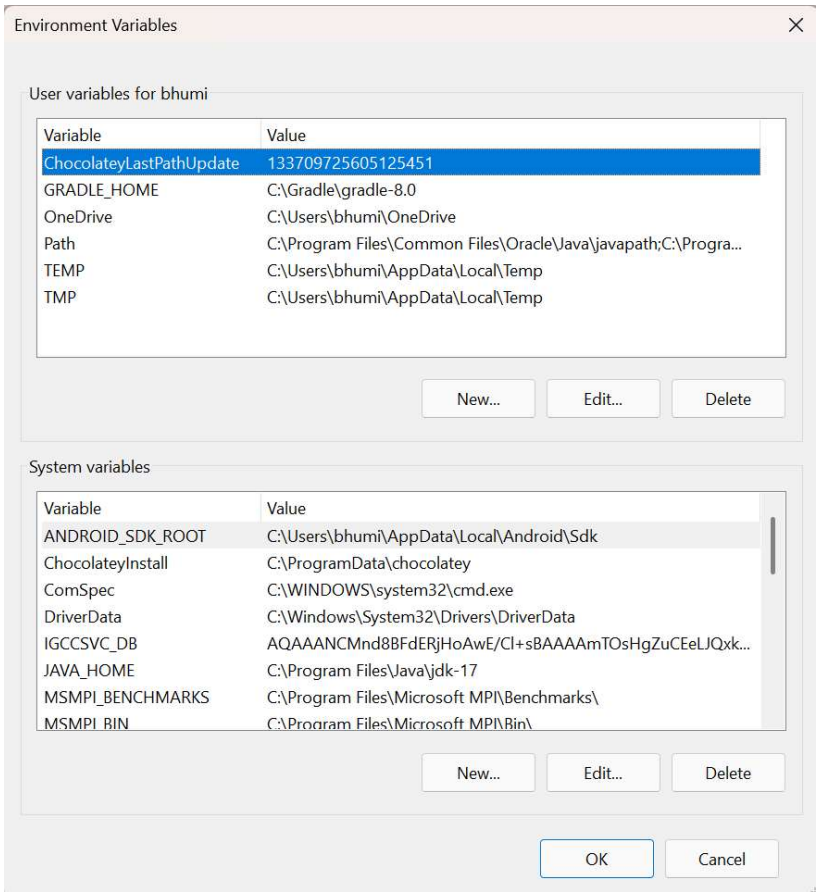



**Step 2:** Next, to download the latest Flutter SDK, click on the Windows **icon**. Here, you will find the download link for SDK.

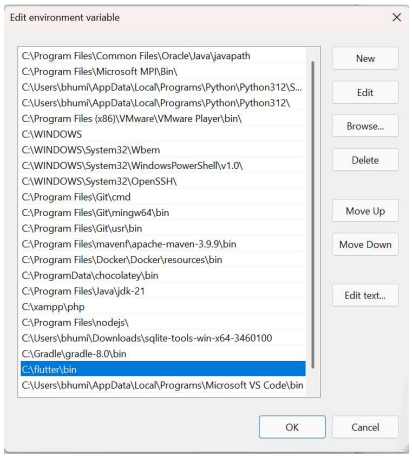
**Step 3:** When your download is complete, extract the **zip** file and place it in the desired installation folder or location, for example, C: /Flutter.

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

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




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




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

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

```
C:\Windows\System32>flutter
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 addition
                            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:
```

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.

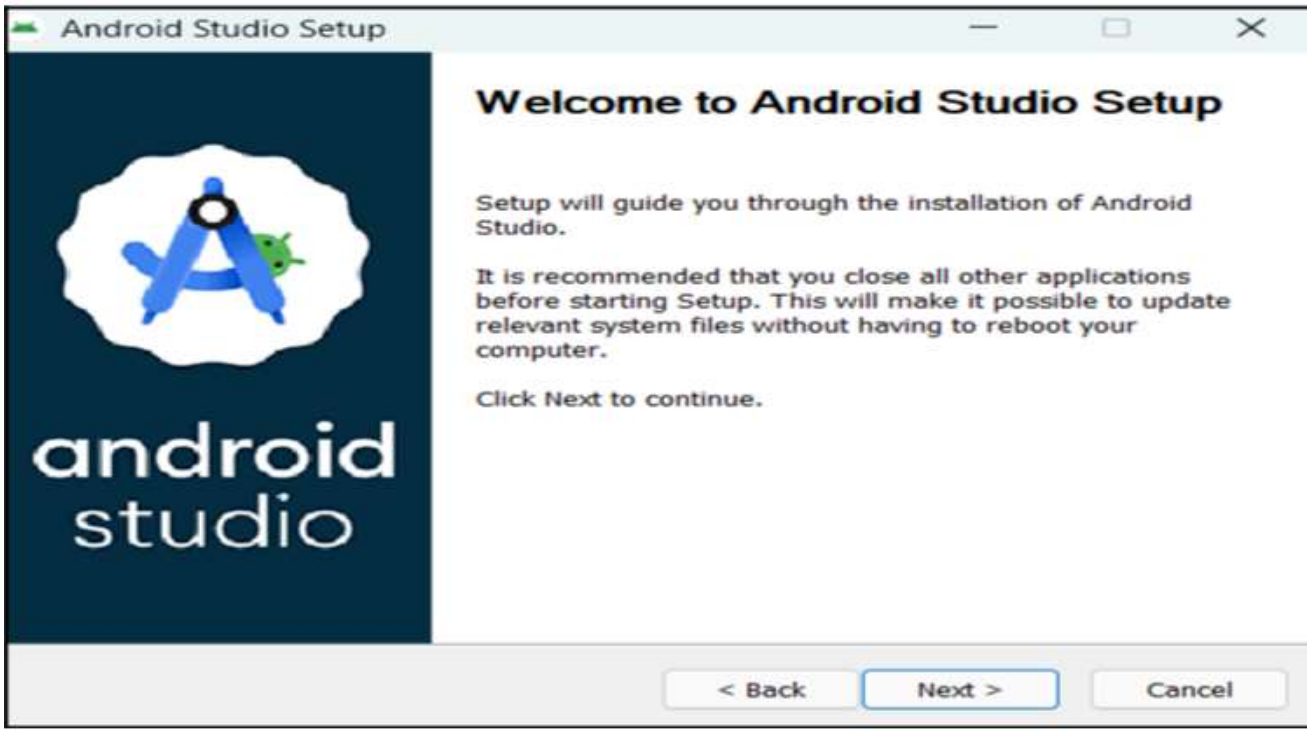
```
C:\Windows\System32>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[√] Flutter (Channel stable, 3.29.2, on Microsoft Windows [Version 10.0.26100.3476], locale en-IN)
[√] Windows Version (11 Home Single Language 64-bit, 24H2, 2009)
[√] Android toolchain - develop for Android devices (Android SDK version 35.0.1)
[√] Chrome - develop for the web
[√] Visual Studio - develop Windows apps (Visual Studio Build Tools 2019 16.11.44)
[√] Android Studio (version 2024.2)
[√] VS Code (version 1.98.2)
[√] Connected device (3 available)
[√] Network resources

• No issues found!
```

**Step 6:** When you run the above command, it will analyze the system and show its report, as shown in the below image. Here, you will find the details of all missing tools, which required to run Flutter as well as the development tools that are available but not connected with the device.

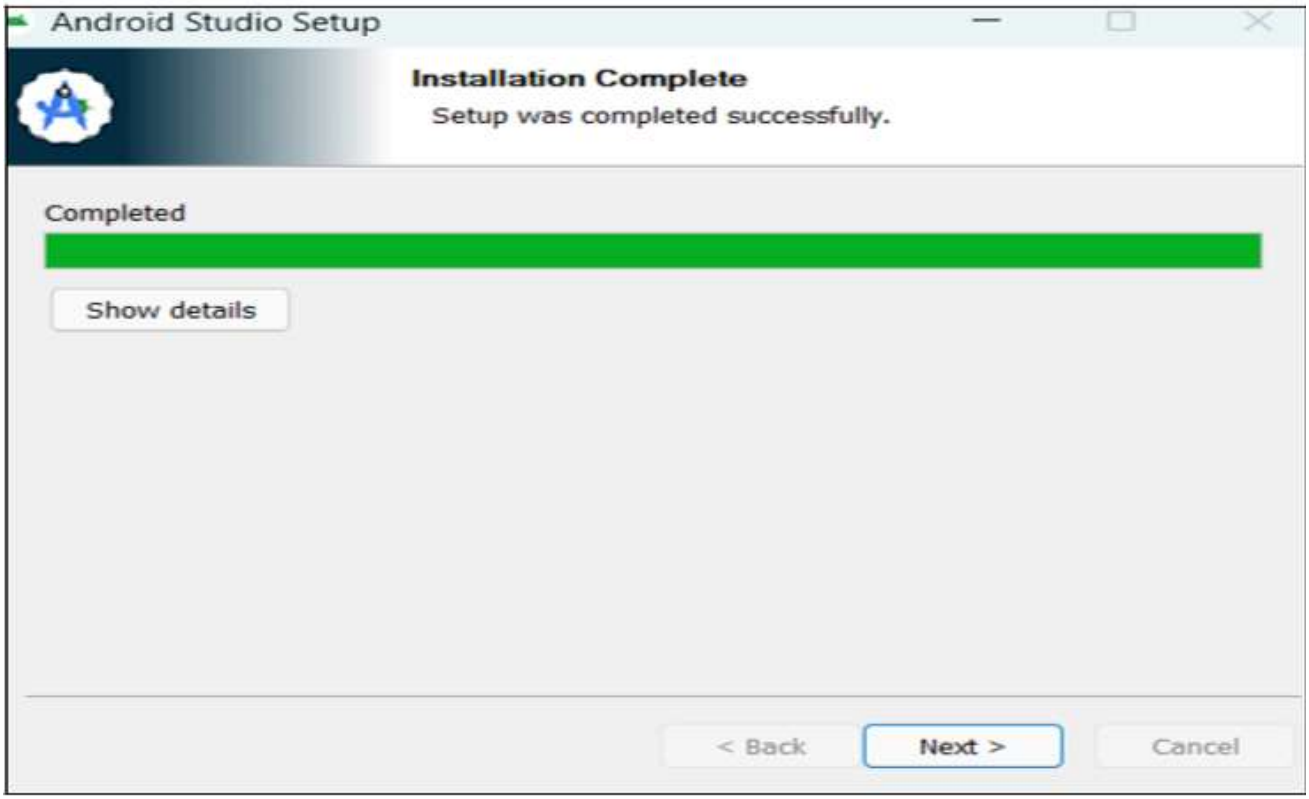
**Step 7:** Install the Android SDK. If the flutter doctor command does not find the Android SDK tool in your system, then you need first to install the Android Studio IDE. To install Android Studio IDE, do the following steps.

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

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

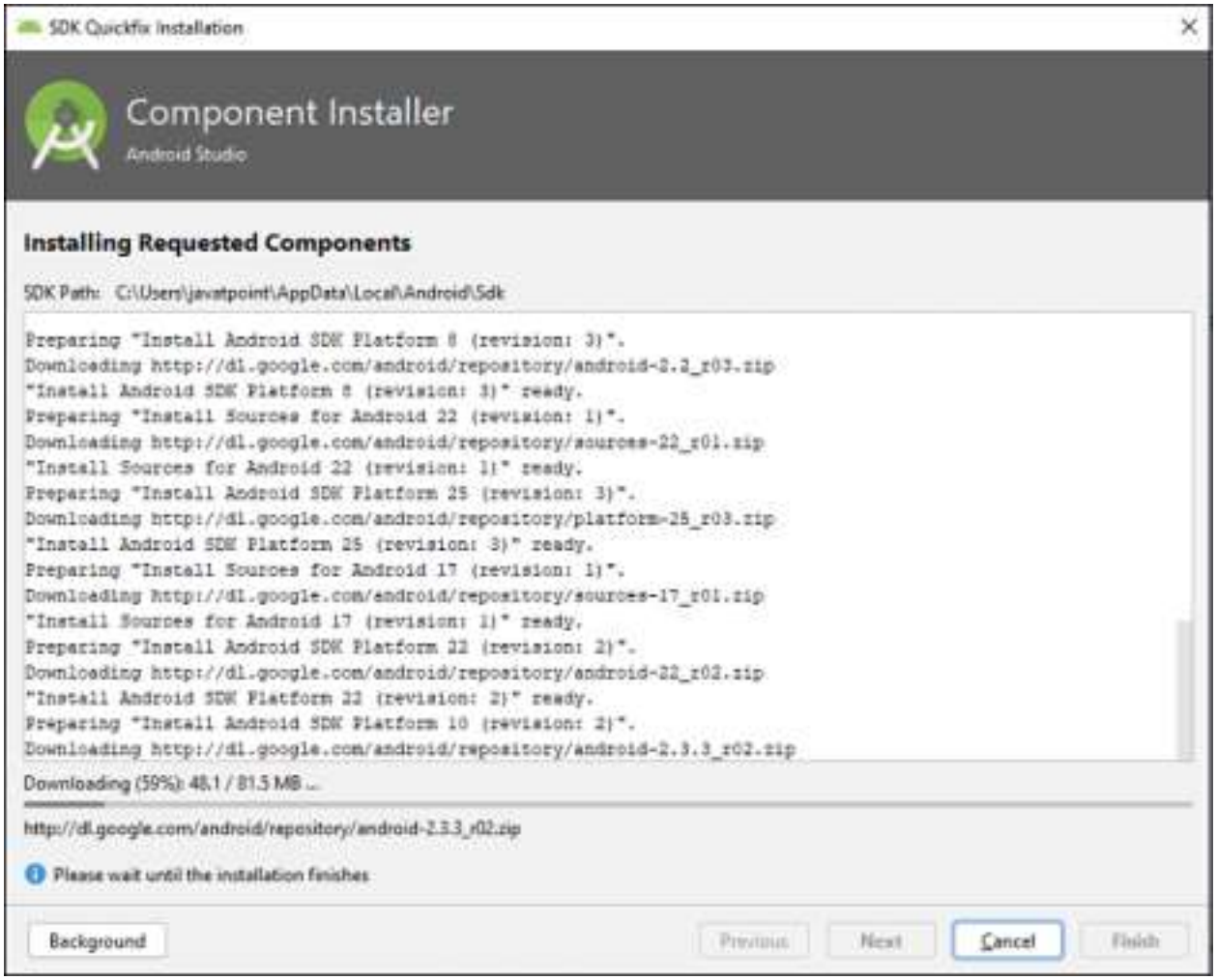



**Step 7.3:** Follow the steps of the installation wizard. Once the installation wizard completes, you will get the following screen.




**Step 7.4:** In the above screen, click Next-> Finish. Once the Finish button is clicked, you need to

choose the 'Don't import Settings option' and click OK. It will start the Android Studio.




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

```
C:\Windows\System32>flutter doctor --android-licenses
Warning: Observed package id 'platform-tools' in inconsistent location 'C:\Users\bhumi\AppData\Local\Android\Sdk\platfor
m-tools.backup' (Expected 'C:\Users\bhumi\AppData\Local\Android\Sdk\platform-tools')
Warning: Already observed package id 'platform-tools' in 'C:\Users\bhumi\AppData\Local\Android\Sdk\platform-tools'. Skip
ping duplicate at 'C:\Users\bhumi\AppData\Local\Android\Sdk\platform-tools.backup'
Warning: Observed package id 'platform-tools' in inconsistent location 'C:\Users\bhumi\AppData\Local\Android\Sdk\platfor
m-tools.backup' (Expected 'C:\Users\bhumi\AppData\Local\Android\Sdk\platform-tools')
Warning: Already observed package id 'platform-tools' in 'C:\Users\bhumi\AppData\Local\Android\Sdk\platform-tools'. Skip
ping duplicate at 'C:\Users\bhumi\AppData\Local\Android\Sdk\platform-tools.backup'
[======================================] 100% Computing updates...
All SDK package licenses accepted.
```

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

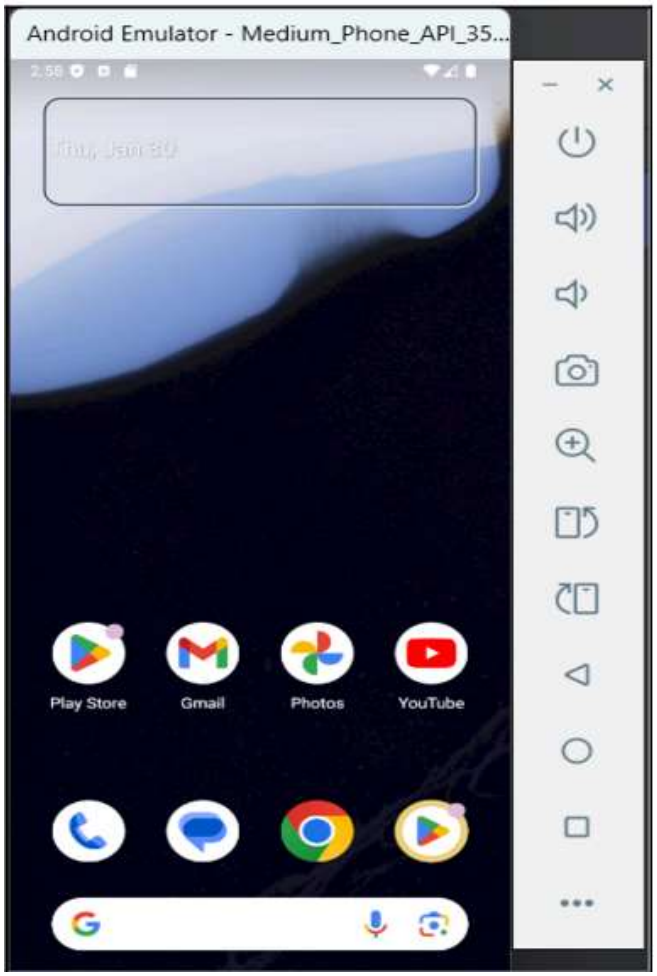
**Step 8.1:** To set an Android emulator, go to Android Studio > Tools > Android > AVD Manager and select Create Virtual Device. Or, go to Help->Find Action->Type Emulator in the search box. You will get the following screen.


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

**Step 8.3:** Select the system image for the latest Android version and click on Next.

**Step 8.4:** Now, verify the all AVD configuration. If it is correct, click on Finish. The following screen appears.


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




**Step 9:** Now, install the Flutter and Dart plugin for building Flutter application in Android Studio. These plugins provide a template to create a Flutter application, give an option to run and debug Flutter application in the Android Studio itself. Do the following steps to install these plugins.

**Step 9.1:** Open the Android Studio and then go to File->Settings->Plugins.

**Step 9.2:** Now, search the Flutter plugin. If found, select Flutter plugin and click install. When you click on install, it will ask you to install Dart plugin as below screen. Click yes to proceed.


**Step 9.3:** Restart the Android Studio.

**Conclusion**: This experiment demonstrates the complete process of setting up the Flutter development environment. The installation involves multiple components including the Flutter SDK, Android Studio IDE, and plugins that work together to create a functional development environment. The Flutter doctor tool helps identify and fix any missing dependencies. Once properly configured, developers can create Flutter projects and run them on emulators or physical devices, providing a foundation for mobile application development using Flutter's cross-platform capabilities.

ATHARVA PATIL(39)
DIV: D15C BATCH: B

# EXPERIMENT 02

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

## Theory:

Flutter is an open-source UI toolkit by Google used to create natively compiled applications for mobile, web, and desktop from a single codebase. It uses the **Dart** programming language and follows a declarative UI approach, making it easy to build beautiful and highly customizable user interfaces.

### Flutter UI Hierarchy

1. **MaterialApp**: The root of the Flutter application (for Material Design apps).
2. **Scaffold**: Provides the basic structure, including AppBar, Body, FloatingActionButton, Bottom Navigation, etc.
3. **Widgets**: The building blocks of the UI.

### Types of Widgets

Widgets in Flutter are categorized into two types:

#### 1. Stateless Widgets

- Immutable (does not change once created).
- Used when the UI does not need to update dynamically.

#### 2. Stateful Widgets

- Can change dynamically during runtime.
- Useful for handling user interactions, animations, and real-time updates.

ATHARVA PATIL(39)
DIV: D15C BATCH: B

## Syntax:

STATELESS WIDGET:

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

STATEFUL WIDGET:

```
class CounterScreen extends StatefulWidget {
  @override
  _CounterScreenState createState() => _CounterScreenState();
}

class _CounterScreenState extends State<CounterScreen> {
  int _counter = 0;

  void _incrementCounter() {
    setState(() {
      _counter++;
    });
  }
```

ATHARVA PATIL(39)
DIV: D15C BATCH: B

## Widget and properties:

### Common widgets

1. Column: The Column widget arranges children vertically in a single column.

Properties of Column

- children → List of widgets to be placed in the column.
- mainAxisAlignment → Aligns children along the vertical axis. (MainAxisAlignment.start, center, end, spaceAround, spaceBetween, spaceEvenly)
- crossAxisAlignment → Aligns children along the horizontal axis. (CrossAxisAlignment.start, center, end, stretch, baseline)
- mainAxisSize → Controls how much space the column should take (MainAxisSize.min or max).
- verticalDirection → Defines the direction in which children are placed (VerticalDirection.down or up).
- textBaseline → Aligns text widgets based on the baseline.

2. Row: The Row widget arranges children horizontally in a single row.

Properties of Row

- children → List of widgets to be placed in the row.
- mainAxisAlignment → Aligns children along the horizontal axis.
- crossAxisAlignment → Aligns children along the vertical axis.
- mainAxisSize → Determines how much space the row should take.
- textBaseline → Aligns text widgets based on the baseline.

3. Stack: The Stack widget places widgets on top of each other in a layered fashion.

Properties of Stack

- children → List of widgets placed in the stack.
- alignment → Aligns children inside the Stack (Alignment.center, topLeft, bottomRight, etc.).
- fit → Controls the size of the non-positioned children (StackFit.loose, StackFit.expand).
- clipBehavior → Defines how content is clipped inside the Stack.

4. Container: The Container widget is a flexible box used to hold and style other widgets.

Properties of Container

- child → The widget inside the container.
- width & height → Dimensions of the container.
- color → Background color of the container.
- alignment → Aligns child inside the container (Alignment.center, topLeft, etc.).
- padding → Space inside the container around its child.
- margin → Space outside the container.
- decoration → Allows adding borders, shadows, gradients, etc.

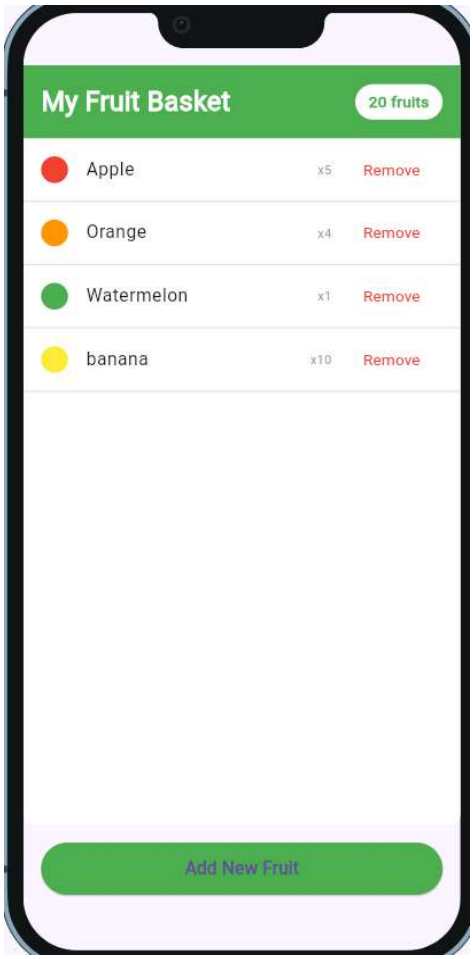
5. List View: The ListView widget displays a scrollable list of items.

Properties of ListView

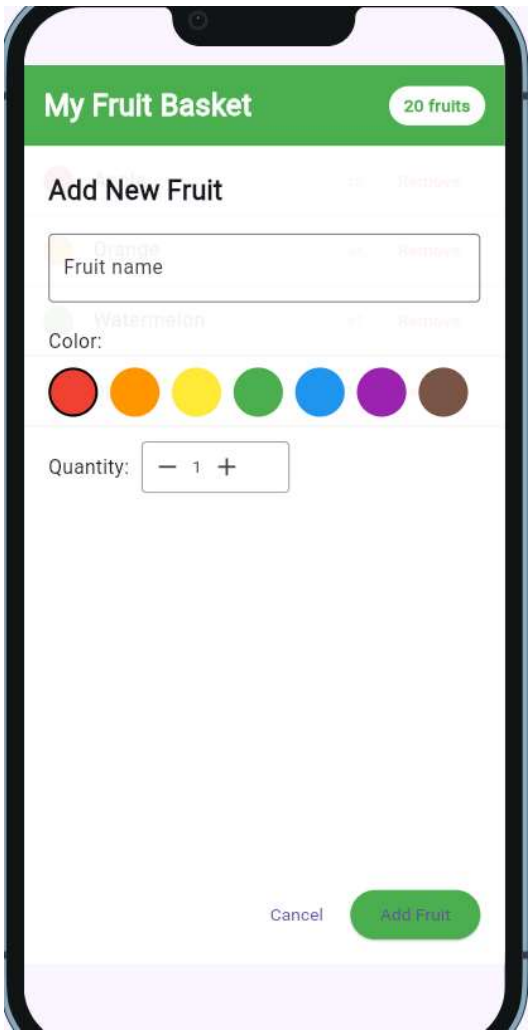
ATHARVA PATIL(39)
DIV: D15C BATCH: B

- children → List of widgets to display in the list (for ListView(children: [...])).
- scrollDirection → Defines the scrolling direction (Axis.vertical or Axis.horizontal).
- shrinkWrap → Adjusts the list size based on children (true or false).
- physics → Defines scrolling behavior (BouncingScrollPhysics, NeverScrollableScrollPhysics).
- padding → Adds padding around the list.
- separatorBuilder → Adds dividers between list items (ListView.separated).
- builder → Dynamically creates items (ListView.builder).

**Output:**



ATHARVA PATIL(39)
DIV: D15C BATCH: B




**Conclusion:** Hence we have successfully designed a flutter UI for creating a fruit basket along with adding and removing fruits and also included common widgets like rows, columns, containers, stack and Lists.

# Experiment No. 3

**AIM :** To include icons, images, fonts in Flutter app

**Theory:**
To include icons, images, and fonts in a Flutter app, you need to understand the following core concepts related to asset management in Flutter. Here's the theory behind including these resources:

1. **Assets in Flutter:**
   Assets are files or resources such as images, fonts, icons, or sounds that you include in your app and bundle within the app package. In Flutter, you can include these assets in your project and then use them in your app.

2. **Adding Assets to pubspec.yaml:**
   In Flutter, you declare assets in the pubspec.yaml file. This is where you specify which assets should be bundled with your app during the build process.
   Example for adding assets:

```
flutter:
  assets:
    - assets/images/
    - assets/icons/
```

   In this example, the images are stored in the assets/images directory, and icons in the assets/icons directory. You can also specify specific files instead of directories.

3. **Including Images:**
   Flutter provides several ways to include images in your app, including network images, asset images, and file images. To use asset images, you reference them by their file path relative to the assets directory.

   Example of including an asset image:

   Image.asset('assets/images/my_image.png')

   For this to work, the image (my_image.png) must be listed in the pubspec.yaml file under the flutter section, like this:

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

**4. Including Icons:**
Flutter allows you to use custom icons in your app. You can add icon files (e.g., .png or .svg) to your assets folder and use them in the app. Alternatively, Flutter provides built-in icons via the Icons class.

Example of using an asset icon:
```
Image.asset('assets/icons/my_icon.png')
```

**5. Including Fonts:**
To include custom fonts, you place your font files (e.g., .ttf or .otf files) in a folder inside your assets directory. Then, you declare these fonts in the pubspec.yaml file and use them in your app.

Example of adding custom fonts in pubspec.yaml:
```
flutter:
  fonts:
    - family: CustomFont
      fonts:
        - asset: assets/fonts/CustomFont-Regular.ttf
        - asset: assets/fonts/CustomFont-Bold.ttf
```

**6. Font Weight and Style:**
When specifying fonts, you can also define specific font weights and styles (like bold, italic) to support different text styles in your app.

**7. Working with Icon Libraries:**
While you can use custom icon files, Flutter also supports popular icon libraries like FontAwesome, MaterialIcons, etc. For example, Flutter's built-in Icons class provides access to the Material Design icons.

Example of using a Material icon:
```
Icon(Icons.home)
```

**8. Caching and Optimization:**
- **Images:** Flutter caches images, but you might want to use libraries like cached_network_image for better image loading and caching.

● **Fonts:** Custom fonts are loaded from assets when the app is first started, and they remain available for the lifecycle of the app.

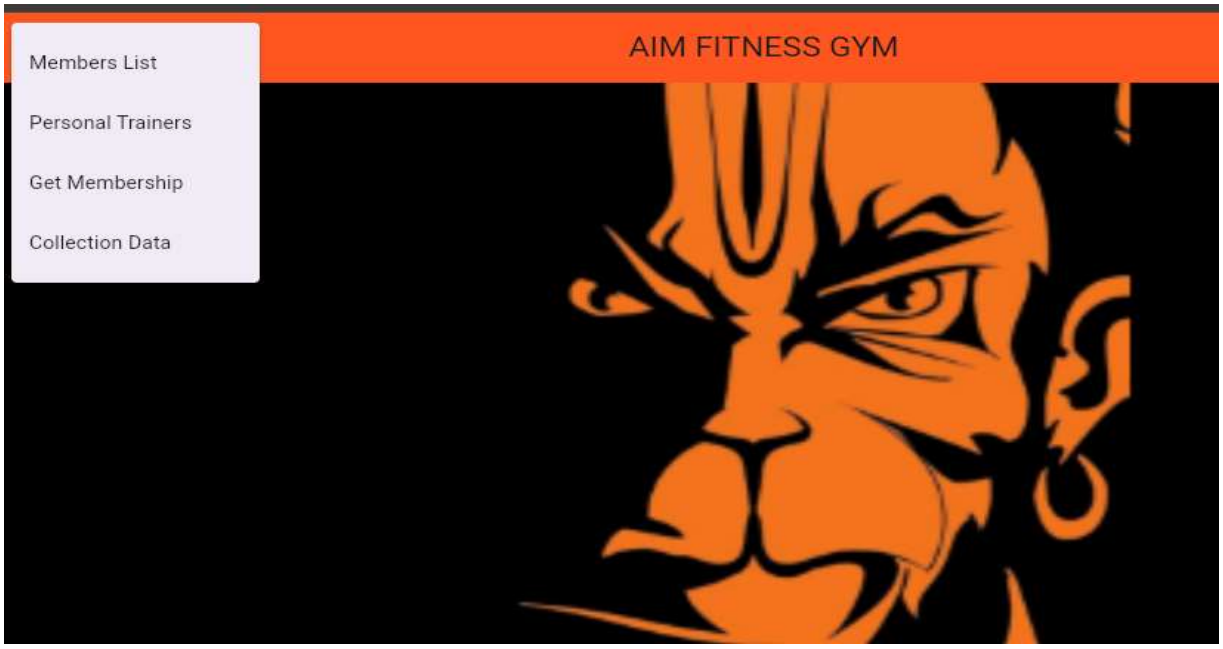
**9. SVG Images:**

If you want to use vector-based images (like SVG), you can include them using packages like flutter_svg, which allows you to display scalable vector graphics (SVG files) in Flutter.

Example of including an SVG:

```
import 'package:flutter_svg/flutter_svg.dart';
SvgPicture.asset('assets/icons/my_icon.svg')
```

**Screenshots**:




**Code Snippets**:

Fonts:

```
Text(
  userData!["name"] ?? "Unknown User",
  style: const TextStyle(
    fontSize: 24,
```

```
    fontWeight: FontWeight.bold,
    color: Color(0xFF3E2723),
  ),
),


title: const Text(
  "Profile",
  style: TextStyle(
    fontSize: 22,
    fontWeight: FontWeight.bold,
  ),
),
```

Icons:

```
leading: const Icon(
  Icons.book,
  color: Color(0xFF04511A),
),
```

Image:

```
CircleAvatar(
  radius: 40,
  backgroundImage: userData!["profilePic"] != null
      ? NetworkImage(userData!["profilePic"])
      : const AssetImage("assets/media/default_avatar.png") as ImageProvider,
),
```

**Conclusion:**

In conclusion, incorporating icons, images, and fonts into a Flutter app significantly enhances the user interface and user experience. By correctly managing assets through the pubspec.yaml file and understanding how to use built-in and custom resources, developers can build visually appealing and functional applications. Whether using asset images, network images, Material icons, SVGs, or custom fonts, Flutter provides flexible and efficient ways to integrate these elements. Mastery of these features empowers developers to design clean, modern, and engaging mobile applications that align with branding and usability standards.

# EXPERIMENT 4

**AIM** : To create an interactive Form using a form widget.

**1. Form Widget:**

- The Form widget is a container for managing form-related interactions. It allows for validation and saving the form data.
- It keeps track of the state of all the fields within the form (like TextFormField widgets) through a GlobalKey<FormState>.

**2. TextFormField:**

- This is the main widget used for collecting user input (such as text). It integrates easily with form validation and submission.
- You can apply validators to the TextFormField to ensure the input meets specific criteria (e.g., required fields, correct format).

**3. GlobalKey:**

- A GlobalKey<FormState> is essential for managing the form's state (e.g., validating fields, saving data). It's assigned to the Form widget and can be used to trigger actions like form validation or saving the data.

**4. Validation:**

- You can define validation rules on each form field. The TextFormField widget has a validator property, which allows you to write logic that will run whenever the form is validated.
- A validator checks whether the input meets the required format (such as checking for valid email format or a non-empty field).

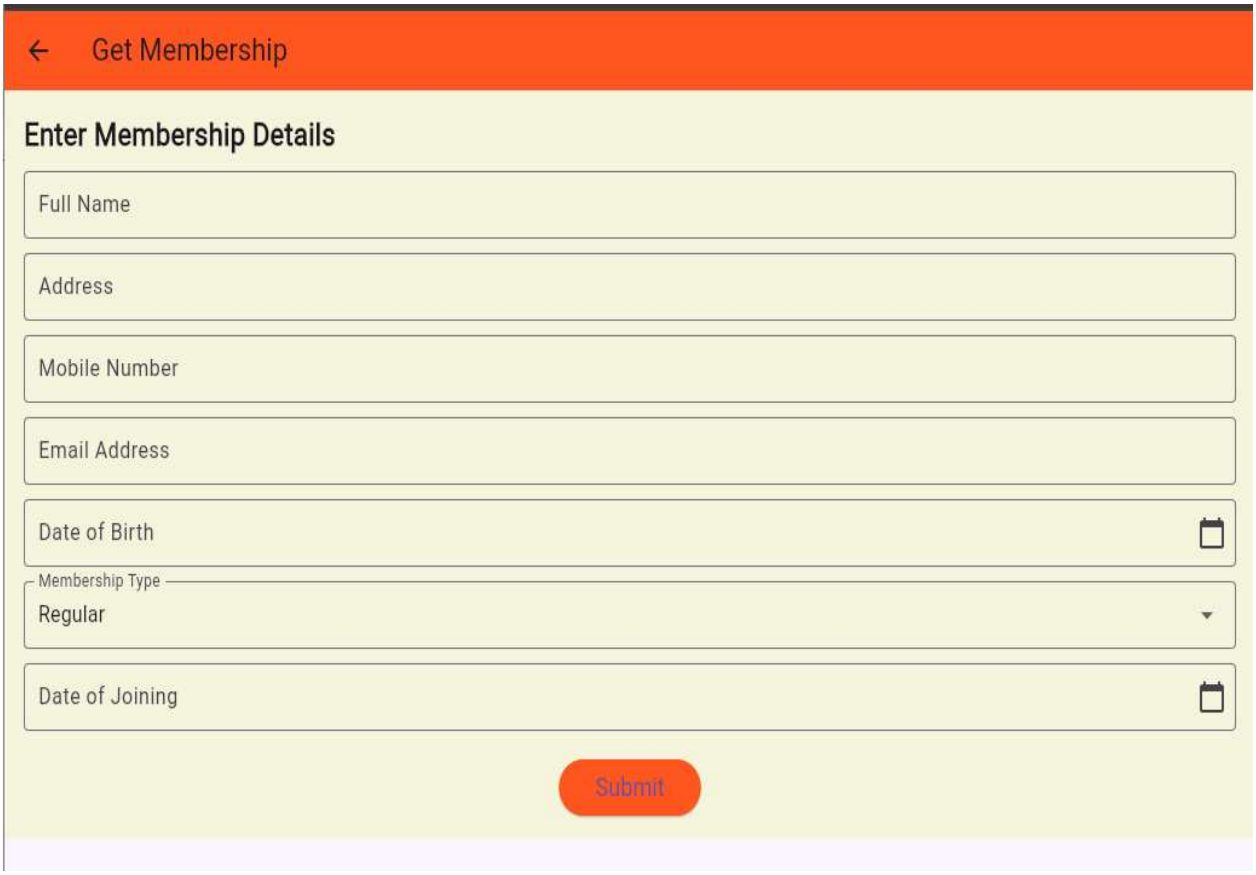
**5. Form Submission:**

- When you're ready to submit the form, you can call formKey.currentState?.save() to trigger the save method for all fields or formKey.currentState?.validate() to check if all the fields pass the validation checks.

**6. Saving Data:**

- After validation, data entered in the form fields can be saved to variables or used for further processing, such as sending it to a server.

**Screenshots:**



**Code**:

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

Form(
  key: _formKey,
  child: Column(
    crossAxisAlignment: CrossAxisAlignment.stretch,
    children: [
      const Text(
        "Create an Account",
        textAlign: TextAlign.center,
        style: TextStyle(
          fontSize: 24,
          fontWeight: FontWeight.bold,
          fontFamily: 'Roboto',
          color: Color(0xFF1E3D34),
        ),
      ),
      const SizedBox(height: 20),

      // Name Field
      TextFormField(
        controller: _nameController,
        decoration: const InputDecoration(
          labelText: "Full Name",
          border: OutlineInputBorder(),
          prefixIcon: Icon(Icons.person, color: Color(0xFF1E3D34)),
        ),
        validator: (value) {
          if (value == null || value.isEmpty) {
            return "Please enter your full name";
          }
          return null;
        },
      ),
      const SizedBox(height: 15),

      // Email Field
      TextFormField(
        controller: _emailController,
```

```
        keyboardType: TextInputType.emailAddress,
        decoration: const InputDecoration(
          labelText: "Email",
          border: OutlineInputBorder(),
          prefixIcon: Icon(Icons.email, color: Color(0xFF1E3D34)),
        ),
        validator: (value) {
          if (value == null || !value.contains('@') || !value.contains('.')) {
            return "Enter a valid email address";
          }
          return null;
        },
      ),
      const SizedBox(height: 15),

      // Password Field
      TextFormField(
        controller: _passwordController,
        obscureText: true,
        decoration: const InputDecoration(
          labelText: "Password",
          border: OutlineInputBorder(),
          prefixIcon: Icon(Icons.lock, color: Color(0xFF1E3D34)),
        ),
        validator: (value) {
          if (value == null ||
!RegExp(r'^(?=.*[A-Z])(?=.*\d)(?=.*[@#\$%^&+=]).{8,}$').hasMatch(value)) {
            return "Password must be 8+ chars, include uppercase, number, and special
char";
          }
          return null;
        },
      ),
      const SizedBox(height: 15),

      // Confirm Password Field
      TextFormField(
        controller: _confirmPasswordController,
        obscureText: true,
        decoration: const InputDecoration(
```

```
          labelText: "Confirm Password",
          border: OutlineInputBorder(),
          prefixIcon: Icon(Icons.lock, color: Color(0xFF1E3D34)),
        ),
        validator: (value) {
          if (value != _passwordController.text) {
            return "Passwords do not match";
          }
          return null;
        },
      ),
    ],
  ),
);
```

**Conclusion**:

In this experiment, we successfully created an interactive form using Flutter's Form and TextFormField widgets. The form included multiple input fields such as full name, email, password, and confirm password. We used a GlobalKey<FormState> to manage the form's state and implemented custom validators for each field to ensure data integrity and proper user input.

The experiment demonstrated how Flutter allows developers to build responsive and user-friendly forms with built-in validation, easy data handling, and clean UI design. This approach is highly efficient for applications that require user registration, login, or any kind of data input.

Overall, this experiment enhanced our understanding of form validation, form state management, and user interaction handling in Flutter.

# EXPERIMENT 5

**AIM**: To apply navigation, routing and gestures in Flutter App.

## Theory

### 1. Navigation and Routing

In Flutter, navigation refers to moving from one screen (or "route") to another. There are several key concepts:

- **Routes**: These are the different screens or pages in your app. Every route is typically represented by a Widget in Flutter. The default route is usually the home screen of the app, but you can define multiple routes for different screens.

- **Navigator**: This is a widget that manages a stack of routes. You can "push" a new route onto the stack to navigate to another screen, or "pop" the top route off to go back.

- **Named Routes**: These are routes that are identified by a string. Instead of pushing or popping routes directly, you can refer to routes by their name (e.g., /home, /settings).

- **Custom Route Transitions**: Flutter allows you to define custom animations and transitions when navigating between routes. You can create smooth, custom page transitions using PageRouteBuilder.

- **Route Arguments**: You can pass data between routes using arguments. This is particularly useful when navigating to a screen that requires specific data (e.g., opening a product page with product details).

### 2. Gestures in Flutter

Gestures are interactions that a user performs with the screen, such as taps, swipes, or long presses. Flutter provides a flexible way to detect these gestures.

- **GestureDetector**: This is the most commonly used widget for detecting gestures. You can wrap it around any widget to detect gestures like tap, double tap, long press, swipe, and others.

- **Tap Gesture**: A simple touch interaction, typically detected using onTap or onLongPress callbacks.

- **Swipe Gestures**: Swiping is usually detected via onHorizontalDragUpdate, onVerticalDragUpdate, or onPanUpdate. These allow you to track the user's finger movement and respond accordingly.

- **Custom Gesture Detection**: Flutter also allows you to implement more complex gestures. For example, you can detect drag gestures to create features like a sliding menu or draggable elements.

- **Dismissible Widget**: This widget enables swipe-to-dismiss behavior, commonly used for items in a list that users can swipe left or right to remove.

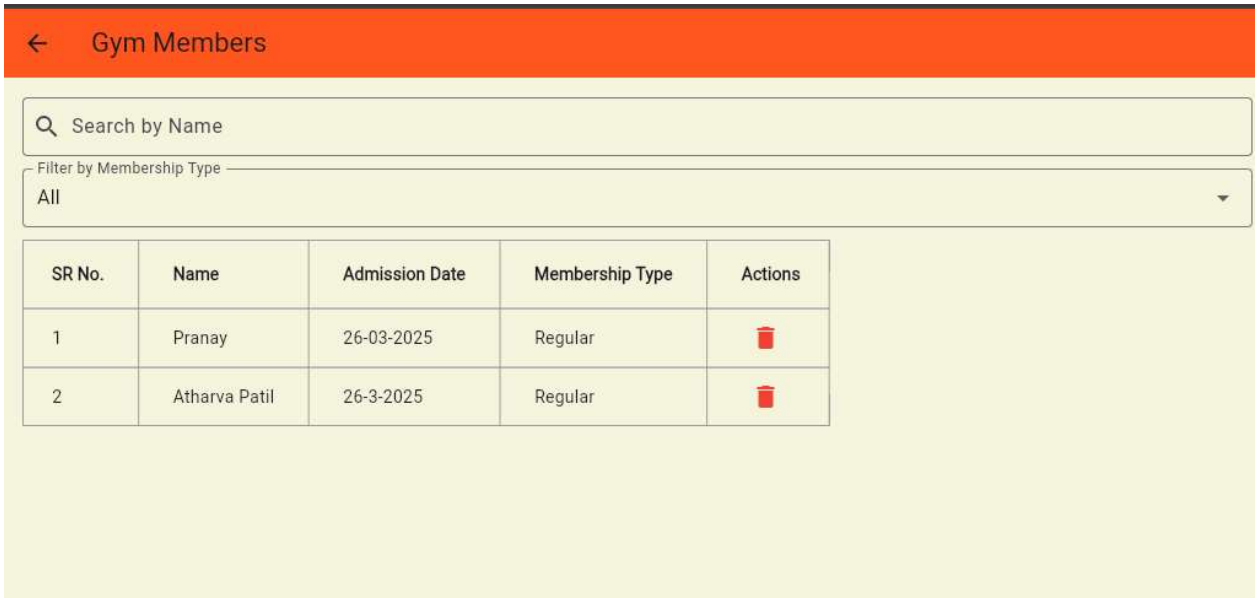
#### 3. Managing Navigation and Gestures Together

When you combine navigation with gestures, you can create more interactive and dynamic UIs. For instance, a user could swipe to navigate between screens, or tap a button that triggers navigation while performing a gesture on a different part of the screen.

#### 4. Back Button Handling

On Android devices, there is a system-wide back button that users can press to navigate backward. Flutter provides a way to intercept and customize this behavior using WillPopScope, allowing you to decide what happens when the user tries to go back (e.g., prevent the user from leaving the current screen, show a confirmation dialog, or allow normal back navigation).

**Screenshots:**




**Code:**

Gesture Detector

```
child: GestureDetector(
  onTap: () => Navigator.push(
    context,
    MaterialPageRoute(builder: (context) => SearchPage()),
  ),
  child: SearchBar(),
),
```

Using Navigator.push

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

```
Navigator.push(
  context,
```

```
  MaterialPageRoute(builder: (context) => const LoginPage()),
);




import 'package:flutter/material.dart';
import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:cached_network_image/cached_network_image.dart';
import 'book_detail_page.dart';

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

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

class _BrowsePageState extends State<BrowsePage> {
  int _selectedIndex = 1;
  String searchQuery = "";
  String? selectedGenre;

  final List<String> genres = [
    "Fiction", "Mystery", "Fantasy", "Sci-Fi", "Romance", "Non-Fiction"
  ];

  void _onItemTapped(int index) {
    switch (index) {
      case 0:
        Navigator.pushNamed(context, '/dashboard');
        break;
      case 1:
        break;
      case 2:
        Navigator.pushNamed(context, '/events');
        break;
      case 3:
        Navigator.pushNamed(context, '/profile');
        break;
```

```
  }
 }

 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(
    title: const Text("Browse Books", style: TextStyle(fontSize: 22, fontWeight:
FontWeight.bold)),
    backgroundColor: const Color(0xFF04511A),
   ),
   body: Padding(
    padding: const EdgeInsets.all(16),
    child: Column(
     crossAxisAlignment: CrossAxisAlignment.start,
     children: [
      // Search Bar
      TextField(
       onChanged: (value) {
        setState(() {
         searchQuery = value.toLowerCase();
        });
       },
       decoration: InputDecoration(
        hintText: "Search books...",
        prefixIcon: const Icon(Icons.search),
        border: OutlineInputBorder(borderRadius: BorderRadius.circular(8)),
       ),
      ),
      const SizedBox(height: 16),

      // Genre Filter
      const Text("Genres", style: TextStyle(fontSize: 18, fontWeight:
FontWeight.bold)),
      const SizedBox(height: 8),
      Wrap(
       spacing: 8,
       children: genres.map((genre) {
        return ChoiceChip(
         label: Text(genre),
```

```
            selected: selectedGenre == genre,
            onSelected: (isSelected) {
              setState(() {
                selectedGenre = isSelected ? genre : null;
              });
            },
          );
        }).toList(),
      ),
      const SizedBox(height: 16),

      // Book List
      const Text("Trending Books", style: TextStyle(fontSize: 18, fontWeight:
FontWeight.bold)),
      const SizedBox(height: 8),
      Expanded(
        child: StreamBuilder(
          stream: FirebaseFirestore.instance.collection('books').snapshots(),
          builder: (context, AsyncSnapshot<QuerySnapshot> snapshot) {
            if (snapshot.connectionState == ConnectionState.waiting) {
              return const Center(child: CircularProgressIndicator());
            }
            if (!snapshot.hasData || snapshot.data!.docs.isEmpty) {
              return const Center(child: Text('No books found.'));
            }

            var books = snapshot.data!.docs.where((doc) {
              var bookData = doc.data() as Map<String, dynamic>;

              // Check if book has a valid cover URL
              if (bookData['coverUrl'] == null || bookData['coverUrl'].toString().isEmpty) {
                return false;
              }

              // Check genre filtering
              if (selectedGenre != null) {
                var bookGenre = bookData['genre'];
                if (bookGenre is String) {
                  if (bookGenre != selectedGenre) return false;
                } else if (bookGenre is List) {
```

```
          if (!bookGenre.contains(selectedGenre)) return false;
        }
      }

      // Search filtering
      if (searchQuery.isNotEmpty) {
        var title = bookData['title']?.toLowerCase() ?? "";
        var author = bookData['author']?.toLowerCase() ?? "";
        if (!title.contains(searchQuery) && !author.contains(searchQuery)) {
          return false;
        }
      }

      return true;
    }).toList();

    if (books.isEmpty) {
      return const Center(child: Text('No books match your filters.'));
    }

    return ListView.builder(
      itemCount: books.length,
      itemBuilder: (context, index) {
        var book = books[index];

        return ListTile(
          contentPadding: const EdgeInsets.symmetric(vertical: 8),
          leading: ClipRRect(
            borderRadius: BorderRadius.circular(8),
            child: CachedNetworkImage(
              imageUrl: book['coverUrl'],
              width: 60,
              height: 90,
              fit: BoxFit.cover,
              placeholder: (context, url) => const Center(child:
CircularProgressIndicator()),
              errorWidget: (context, url, error) => const Icon(Icons.broken_image,
size: 50),
            ),
          ),
```

```
                    title: Text(book['title'], style: const TextStyle(fontWeight:
FontWeight.bold)),
                    subtitle: Text(book['author']),
                    onTap: () {
                      Navigator.push(
                        context,
                        MaterialPageRoute(
                          builder: (context) => BookDetailPage(
                            bookId: book.id,
                            bookTitle: book['title'],
                            bookCover: book['coverUrl'],
                            bookAuthor: book['author'],
                            bookDescription: book['description'],
                          ),
                        ),
                      );
                    },
                  );
                },
              );
            },
          ),
        ),
      ],
    ),
  ),
  bottomNavigationBar: BottomNavigationBar(
    currentIndex: _selectedIndex,
    onTap: _onItemTapped,
    backgroundColor: const Color(0xFF04511A),
    selectedItemColor: Colors.white,
    unselectedItemColor: Colors.white70,
    type: BottomNavigationBarType.fixed,
    items: const [
      BottomNavigationBarItem(icon: Icon(Icons.dashboard), label: 'Home'),
      BottomNavigationBarItem(icon: Icon(Icons.search), label: 'Browse'),
      BottomNavigationBarItem(icon: Icon(Icons.event), label: 'Events'),
      BottomNavigationBarItem(icon: Icon(Icons.person), label: 'Profile'),
    ],
  ),
```

```
    );
  }
}
```

**Conclusion:**

This experiment provided a comprehensive understanding of navigation and routing in Flutter, which is essential for creating multi-screen applications. We learned how to implement smooth transitions between different screens using both direct and named routes. The use of the Navigator widget helped us manage route stacks effectively, improving the overall navigation flow of the app.

Additionally, the integration of gesture detection using widgets like GestureDetector enabled us to build more interactive and responsive user interfaces. By responding to user actions such as taps and swipes, we were able to make the application more dynamic, intuitive, and user-friendly. These features are crucial for enhancing the user experience and building polished, production-ready apps.

Overall, this experiment strengthened our understanding of managing screen navigation and incorporating user gestures, both of which are fundamental for modern mobile app development in Flutter.

# EXPERIMENT 6

**AIM**: To Set Up Firebase with Flutter for iOS and Android Apps.

## Theory

### Introduction to Firebase and Flutter Integration

Firebase is a comprehensive platform developed by Google, designed to help developers build high-quality applications for both mobile and web. It provides essential services such as real-time databases, authentication, cloud storage, hosting, and much more. One of the most widely used Firebase services is the Firebase Realtime Database, which is a NoSQL cloud database that allows data to be stored and synced in real-time across all connected devices.

Flutter, on the other hand, is an open-source UI software development kit created by Google, which allows developers to build natively compiled applications for mobile, web, and desktop from a single codebase. Its rich set of pre-designed widgets and powerful tools makes Flutter an attractive option for developing visually appealing and performant applications.

Integrating Firebase with Flutter allows developers to leverage the full potential of Firebase services in their applications. By using Firebase's Realtime Database, Flutter apps can achieve features such as real-time data synchronization, secure authentication, and cloud-based storage. This combination enables developers to create powerful, scalable, and feature-rich mobile and web applications.

### Firebase Realtime Database Overview

Firebase Realtime Database is a cloud-hosted NoSQL database that stores data in a JSON-like format. The key characteristic of this database is its real-time synchronization feature, meaning that any changes made to the database are instantly reflected on all clients (i.e., devices) connected to it.

This makes it an ideal solution for applications that require frequent updates and need to maintain synchronized data across multiple users or devices, such as messaging apps, social media platforms, or collaborative tools.

The Firebase Realtime Database is structured as a tree of data, where each node in the tree can contain key-value pairs. This structure allows for easy data retrieval and modification. Firebase's real-time capabilities enable apps to immediately receive updates to the data whenever it changes, without the need to refresh or reload the page.

Additionally, the database supports offline data persistence, meaning that even if the user's device loses its internet connection, the app can still function by using the locally cached data.

**Setting Up Firebase in Flutter**

To connect a Flutter app with Firebase, the following steps are typically followed:

**1. Creating a Firebase Project:**

To start using Firebase with Flutter, the first step is to create a Firebase project in the Firebase Console. Once the project is created, developers can associate their Flutter app with the Firebase project by following the platform-specific instructions for Android or iOS. This usually involves configuring API keys, downloading configuration files, and adding them to the Flutter project.

**2. Integrating Firebase SDK in Flutter:**

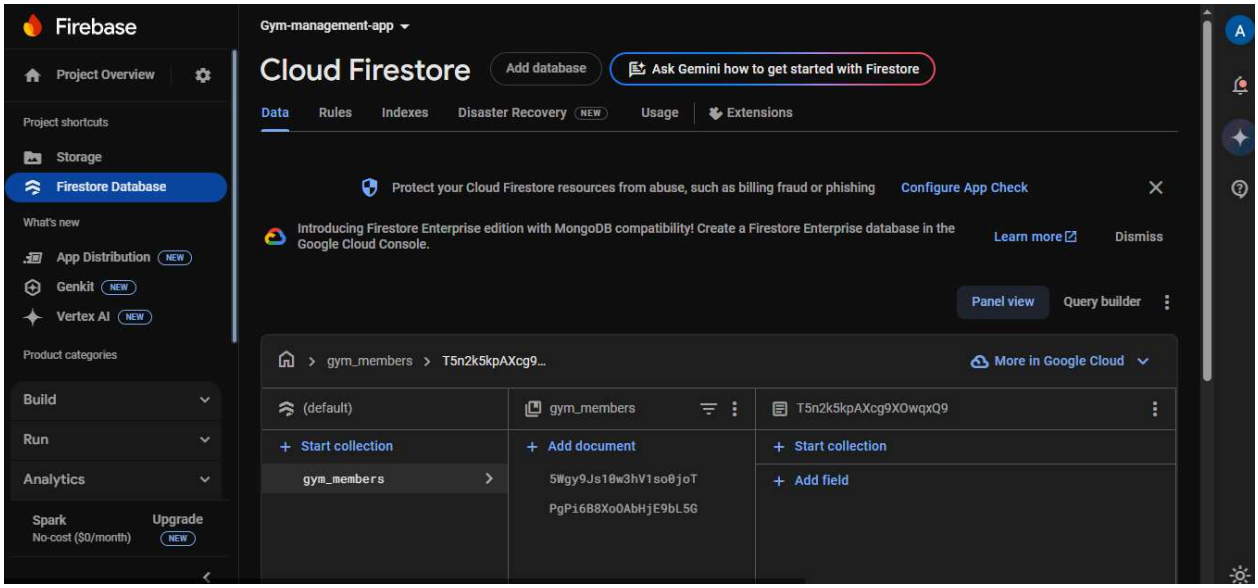
After the Firebase project is set up, developers need to integrate Firebase's SDK into the Flutter app. This involves adding the necessary dependencies to the Flutter project's pubspec.yaml file. For Firebase's Realtime Database, the package firebase_database is used. Additionally, Firebase's core SDK (firebase_core) must also be included to initialize Firebase services.

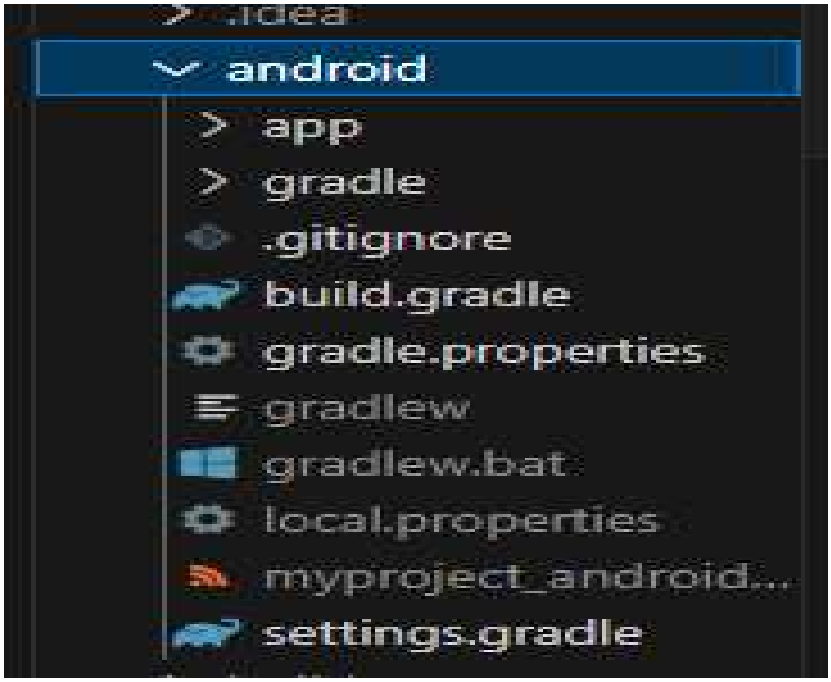
**3. Initializing Firebase:**

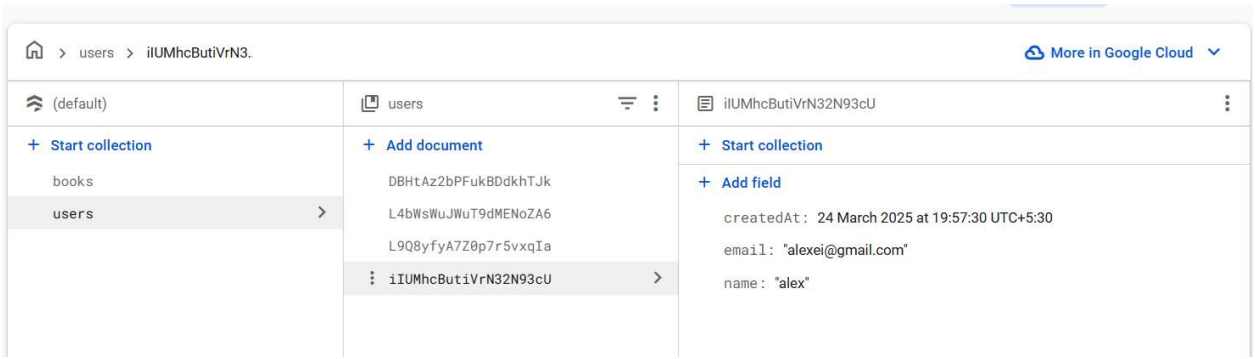
Before any Firebase functionality can be used, it is essential to initialize Firebase in the Flutter app. This is done by calling Firebase.initializeApp() in the main entry point of the app (usually in the main.dart file). Firebase needs to be initialized before interacting with any Firebase services, such as the Realtime Database, Cloud Firestore, or Authentication.

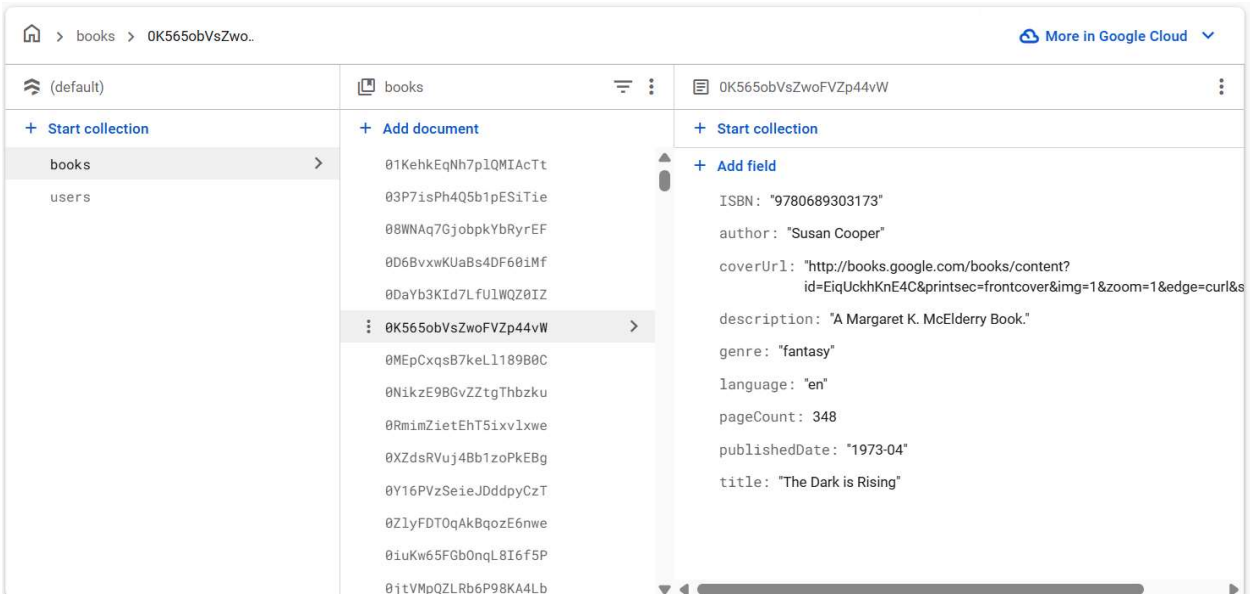
Connecting Firebase to a Flutter app enables developers to create robust, scalable, and real-time applications with ease. Firebase's Realtime Database offers a powerful, cloud-based solution for managing data in real-time, while Firebase Authentication ensures secure access control. By integrating Firebase with Flutter, developers can take advantage of real-time data synchronization, offline support, and a wide range of other Firebase features, allowing them to build feature-rich apps that meet modern user expectations.

## Screenshots:






**Code**:

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

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

  @override
  _SignUpPageState createState() => _SignUpPageState();
}
```

```
class _SignUpPageState extends State<SignUpPage> {
  final _formKey = GlobalKey<FormState>();
  final TextEditingController _nameController = TextEditingController();
  final TextEditingController _emailController = TextEditingController();
  final TextEditingController _passwordController = TextEditingController();
  final TextEditingController _confirmPasswordController = TextEditingController();

  bool _isLoading = false;
  String? _errorMessage;

  @override
  void dispose() {
    _nameController.dispose();
    _emailController.dispose();
    _passwordController.dispose();
    _confirmPasswordController.dispose();
    super.dispose();
  }

  Future<void> _signUp() async {
    if (!_formKey.currentState!.validate()) return;

    setState(() {
      _isLoading = true;
      _errorMessage = null;
    });

    try {
      String name = _nameController.text.trim();
      String email = _emailController.text.trim().toLowerCase();
      String password = _passwordController.text.trim();

      // Create user with Firebase Authentication
      UserCredential userCredential = await
FirebaseAuth.instance.createUserWithEmailAndPassword(
        email: email,
        password: password,
      );
```

```
  User? user = userCredential.user;

  if (user != null) {
    // Save additional user details in Firestore
    await FirebaseFirestore.instance.collection('users').doc(user.uid).set({
      'name': name,
      'email': email,
      'createdAt': FieldValue.serverTimestamp(),
    });

    // Save login info to SharedPreferences
    SharedPreferences prefs = await SharedPreferences.getInstance();
    await prefs.setString('loggedInUser', email);
    await prefs.setString('userId', user.uid);

    // Success message and navigation
    ScaffoldMessenger.of(context).showSnackBar(
      const SnackBar(content: Text("Account created successfully!")),
    );

    Navigator.pushReplacementNamed(context, '/dashboard');
  }
} on FirebaseAuthException catch (e) {
  setState(() {
    if (e.code == 'email-already-in-use') {
      _errorMessage = "Email is already registered. Please log in.";
    } else if (e.code == 'weak-password') {
      _errorMessage = "Password is too weak.";
    } else {
      _errorMessage = "Failed to sign up. ${e.message}";
    }
  });
} catch (e) {
  setState(() {
    _errorMessage = "Something went wrong. Please try again.";
  });
} finally {
  setState(() {
    _isLoading = false;
  });
```

```
  }
 }

 @override
 Widget build(BuildContext context) {
  return Scaffold(
   backgroundColor: const Color(0xFF1E3D34),
   appBar: AppBar(
    title: const Text("Sign Up"),
    backgroundColor: const Color(0xFF1E3D34),
   ),
   body: Center(
    child: SingleChildScrollView(
     child: Padding(
      padding: const EdgeInsets.all(20),
      child: Container(
       width: MediaQuery.of(context).size.width * 0.85,
       padding: const EdgeInsets.all(20),
       decoration: BoxDecoration(
        color: Colors.white,
        borderRadius: BorderRadius.circular(20),
        border: Border.all(color: const Color(0xFF1E3D34), width: 2),
        boxShadow: const [BoxShadow(color: Colors.black26, blurRadius: 6,
spreadRadius: 2)],
       ),
       child: Form(
        key: _formKey,
        child: Column(
         crossAxisAlignment: CrossAxisAlignment.stretch,
         children: [
          const Text(
           "Create an Account",
           textAlign: TextAlign.center,
           style: TextStyle(
            fontSize: 24,
            fontWeight: FontWeight.bold,
            fontFamily: 'Roboto',
            color: Color(0xFF1E3D34),
           ),
          ),
```

```
                const SizedBox(height: 20),

                TextFormField(
                  controller: _nameController,
                  decoration: const InputDecoration(
                    labelText: "Full Name",
                    border: OutlineInputBorder(),
                    prefixIcon: Icon(Icons.person, color: Color(0xFF1E3D34)),
                  ),
                  validator: (value) => value == null || value.isEmpty ? "Please enter your
full name" : null,
                ),
                const SizedBox(height: 15),

                TextFormField(
                  controller: _emailController,
                  keyboardType: TextInputType.emailAddress,
                  decoration: const InputDecoration(
                    labelText: "Email",
                    border: OutlineInputBorder(),
                    prefixIcon: Icon(Icons.email, color: Color(0xFF1E3D34)),
                  ),
                  validator: (value) =>
                      value == null || !value.contains('@') || !value.contains('.') ? "Enter a valid
email" : null,
                ),
                const SizedBox(height: 15),

                TextFormField(
                  controller: _passwordController,
                  obscureText: true,
                  decoration: const InputDecoration(
                    labelText: "Password",
                    border: OutlineInputBorder(),
                    prefixIcon: Icon(Icons.lock, color: Color(0xFF1E3D34)),
                  ),
                  validator: (value) {
                    if (value == null ||

!RegExp(r'^(?=.*[A-Z])(?=.*\d)(?=.*[@#\$%^&+=]).{8,}$').hasMatch(value)) {
```

```
                    return "Password must be 8+ chars, include uppercase, number, and special char";
                  }
                  return null;
                },
              ),
              const SizedBox(height: 15),

              TextFormField(
                controller: _confirmPasswordController,
                obscureText: true,
                decoration: const InputDecoration(
                  labelText: "Confirm Password",
                  border: OutlineInputBorder(),
                  prefixIcon: Icon(Icons.lock, color: Color(0xFF1E3D34)),
                ),
                validator: (value) => value != _passwordController.text ? "Passwords do not match" : null,
              ),
              const SizedBox(height: 15),

              if (_errorMessage != null)
                Padding(
                  padding: const EdgeInsets.symmetric(vertical: 8),
                  child: Text(
                    _errorMessage!,
                    style: const TextStyle(color: Colors.red, fontSize: 14),
                    textAlign: TextAlign.center,
                  ),
                ),

              ElevatedButton(
                style: ElevatedButton.styleFrom(
                  backgroundColor: const Color(0xFF1E3D34),
                  padding: const EdgeInsets.symmetric(vertical: 14),
                  shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(10)),
                ),
                onPressed: _isLoading ? null : _signUp,
                child: _isLoading
```

```
                    ? const CircularProgressIndicator(color: Colors.white)
                    : const Text("SIGN UP", style: TextStyle(color: Colors.white, fontWeight:
FontWeight.bold)),
                ),
                const SizedBox(height: 15),

                TextButton(
                  onPressed: () => Navigator.pushNamed(context, '/login'),
                  child: const Text(
                    "Already have an account? Log in",
                    style: TextStyle(color: Color(0xFF1E3D34)),
                  ),
                ),
              ],
            ),
          ),
        ),
      ),
    ),
  );
 }
}
```

**Conclusion**:
This experiment provided comprehensive, hands-on experience in integrating Firebase services into a Flutter application for both Android and iOS platforms. By incorporating Firebase Authentication, Firestore, and other cloud-based services, it enabled the development of a robust backend system that supports secure user registration, data storage, and real-time data updates. The integration allowed the app to benefit from Firebase's powerful and scalable infrastructure, which is essential for building modern, responsive, and data-driven mobile applications. Additionally, working with Firebase helped in understanding how cloud-based services can enhance the overall functionality, reliability, and user experience of cross-platform mobile apps.

# EXPERIMENT NO. 7

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

Theory:-

- 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

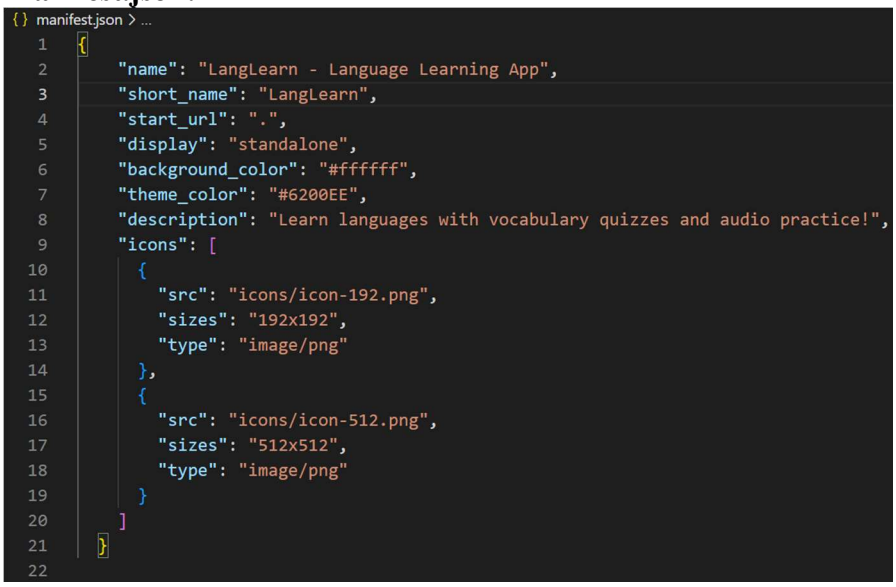
Progressive web apps can be installed on the user device like a native device, but it does not demand submission on an App Store. This makes it far more cost-effective than native mobile applications while offering the same set of functionalities.

The main features are:

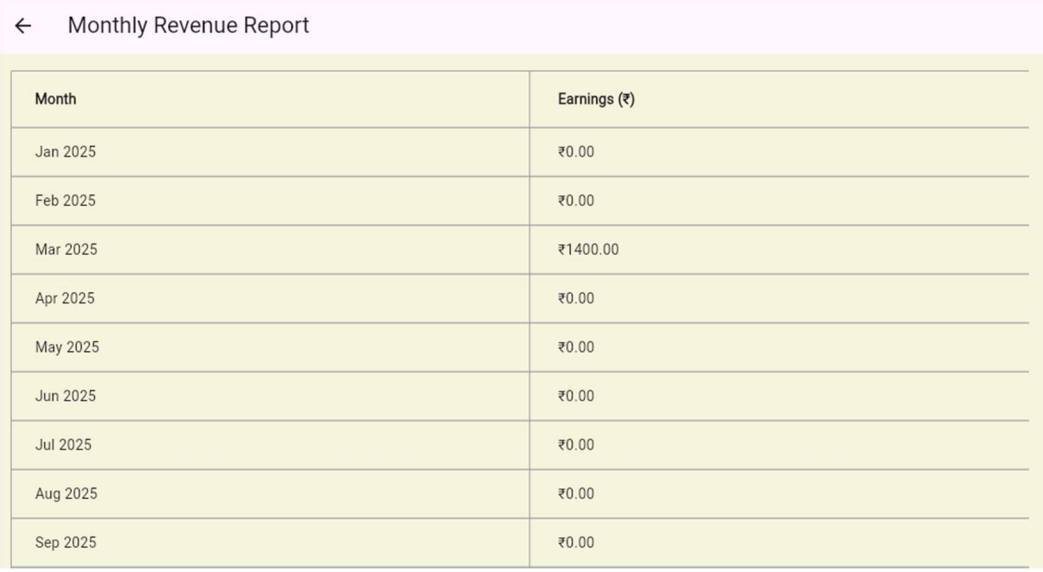
1. Progressive - They work for every user, regardless of the browser chosen because they are built at the base with progressive improvement principles.
2. Responsive - They adapt to the various screen sizes: desktop, mobile, tablet, or dimensions that can later become available.
3. App-like - They behave with the user as if they were native apps, in terms of interaction and navigation.
4. Updated - Information is always up-to-date thanks to the data update process offered by service workers.
5. Secure - Exposed over HTTPS protocol to prevent the connection from displaying information or altering the contents.
6. Searchable - They are identified as "applications" and are indexed by search engines.
7. Reactivable - Make it easy to reactivate the application thanks to capabilities such as web notifications.
8. 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.
9. Linkable - Easily shared via URL without complex installations.
10. 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.

**Code:**

**Manifest.json:**



**Output:**




**Conclusion:**

In this experiment, metadata was successfully added to the PWA to enhance its visibility, branding, and user experience. The use of manifest.json and HTML <meta> tags allowed the app to behave like a native application, support SEO, and display correctly when shared on social media. This setup ensures the PWA is well-optimized and user-friendly across various platforms.

# Experiment No. 8

**Aim**: To code and register a service worker, and complete the install and activation process for a new service worker for the E-commerce PWA.

**Theory**:

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

Things to note about Service Worker:

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

What can we do with Service Workers?

- You can dominate **Network Traffic**
  You can manage all network traffic of the page and do any manipulations. For example, when the page requests a CSS file, you can send plain text as a response or when the page requests an HTML file, you can send a png file as a response. You can also send a true response too.

- You can **Cache**
  You can cache any request/response pair with Service Worker and Cache API and you can access these offline content anytime.

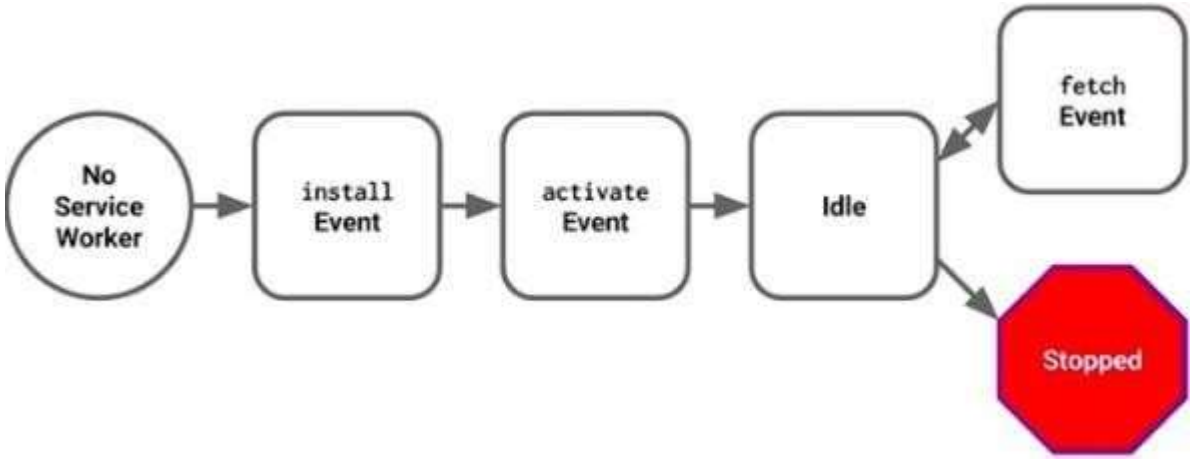
- You can manage **Push Notifications**
  You can manage push notifications with Service Worker and show any information message to the user.
- You can **Continue**
  Although Internet connection is broken, you can start any process with Background Sync of Service Worker.

What can't we do with Service Workers?

- You can't access the **Window**
  You can't access the window, therefore, You can't manipulate DOM elements. But, you can communicate to the window through post Message and manage processes that you want.

- You can't work it on **80 Port**
  Service Worker just can work on HTTPS protocol. But you can work on localhost during development.

Service Worker Cycle




A service worker goes through three steps in its life cycle:

- Registration
- Installation
- Activation

Registration
To install a service worker, you need to register it in your main JavaScript code. Registration tells the browser where your service worker is located, and to start installing it in the background. Let's look at an example:

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

This code starts by checking for browser support by examining **navigator.serviceWorker**. The service worker is then registered with navigator.serviceWorker.register, which returns a promise that resolves when the service worker has been successfully registered. The scope of the service worker is then logged with registration.scope. If the service worker is already installed, navigator.serviceWorker.register returns the registration object of the currently active service worker.

The scope of the service worker determines which files the service worker controls, in other words, from which path the service worker will intercept requests. The default scope is the location of the service worker file, and extends to all directories below. So if service-worker.js is located in the root directory, the service worker will control requests from all files at this domain.

You can also set an arbitrary scope by passing in an additional parameter when registering. For

example: main.js

```
navigator.serviceWorker.register('/service-worker.js', { scope: '/app/'
});
```

In this case we are setting the scope of the service worker to /app/, which means the service worker will control requests from pages like /app/, /app/lower/ and /app/lower/lower, but not from pages like /app or /, which are higher.

If you want the service worker to control higher pages e.g. /app (without the trailing slash) you can indeed change the scope option, but you'll also need to set the Service-Worker-Allowed HTTP Header in your server config for the request serving the service worker script.

```
navigator.serviceWorker.register('/app/service-worker.js', { scope: '/app'
});
```

Installation

Once the browser registers a service worker, installation can be attempted. This occurs if the service worker is considered to be new by the browser, either because the site currently doesn't have a registered service worker, or because there is a byte difference between the new service worker and the previously installed one.

A service worker installation triggers an install event in the installing service worker. We can include an install event listener in the service worker to perform some task when the service worker installs. For instance, during the install, service workers can precache parts of a web app so that it loads instantly the next time a user opens it (see caching the application shell). So, after that first load, you're going to benefit from instant repeat loads and your time to interactivity is going to be even better in those cases. An example of an installation event listener looks like this:

service-worker.js

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

Activation

Once a service worker has successfully installed, it transitions into the activation stage. If there are any open pages controlled by the previous service worker, the new service worker enters a waiting state. The new service worker only activates when there are no longer any pages loaded that are still using the old service worker. This ensures that only one version of the service worker is running at any given time.

When the new service worker activates, an activate event is triggered in the activating service worker. This event listener is a good place to clean up outdated caches (see the Offline Cookbook for an example).

service-worker.js

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

Once activated, the service worker controls all pages that load within its scope, and starts listening for events from those pages. However, pages in your app that were loaded before the service worker activation will not be under service worker control. The new service worker will only take over when you close and reopen your app, or if the service worker calls **clients.claim()**. Until then, requests from this page will not be intercepted by the new service worker. This is intentional as a way to ensure consistency in your site.

**Code:**

```
<script>
        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);
              });
          });
        }

</script>
```

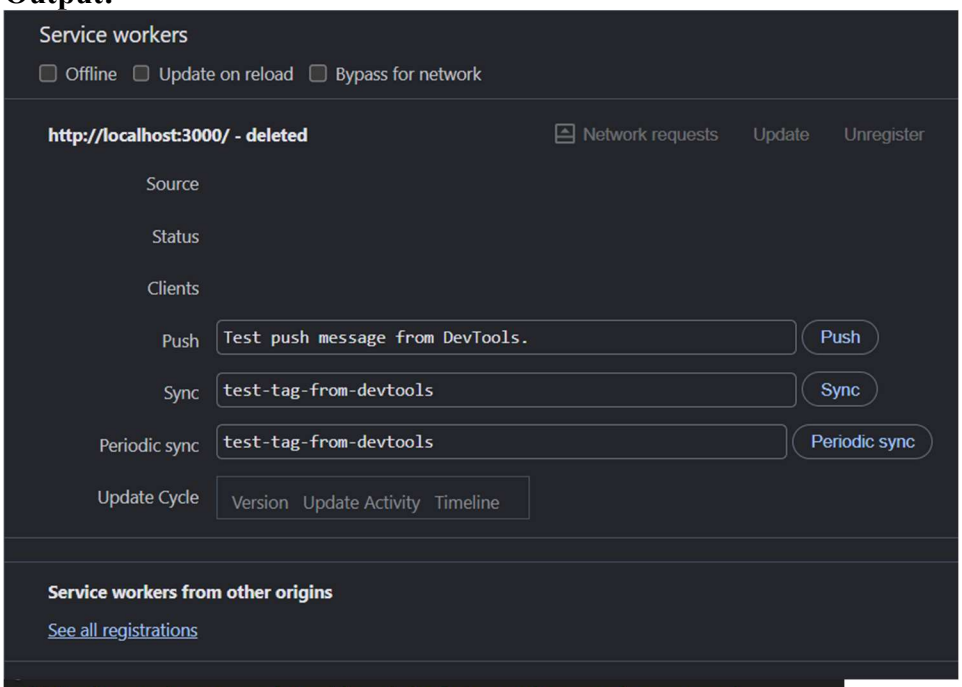
**service-worker.js**

```
const CACHE_NAME = 'my-pwa-cache-v1';
const urlsToCache = [
  '/',
  '/index.html',
  '/styles.css',
  '/icons/icon-192x192.png',
  '/icons/icon-512x512.png',
  '/bgvid.mp4',
  '/media/logo.png',
  '/media/madwoman.png'
];

// Install event: cache essential resources
self.addEventListener('install', (event) => {
  console.log('[Service Worker] Install');
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then((cache) => {
        console.log('[Service Worker] Caching files');
        return cache.addAll(urlsToCache);
      })
      .catch((error) => console.error('[Service Worker] Failed to cache files:', error))
```

```
  );
  self.skipWaiting(); // Activate the new service worker immediately
});

// Activate event: clean up old caches
self.addEventListener('activate', (event) => {
  console.log('[Service Worker] Activate');
  event.waitUntil(
    caches.keys()
      .then((cacheNames) => {
        return Promise.all(
          cacheNames.map((cacheName) => {
            if (cacheName !== CACHE_NAME) {
              console.log('[Service Worker] Deleting old cache:', cacheName);
              return caches.delete(cacheName);
            }
          })
        );
      })
  );
  self.clients.claim(); // Take control of all clients immediately
});

// Fetch event: serve from cache first, then network fallback
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request)
      .then((cachedResponse) => {
        if (cachedResponse) {
          return cachedResponse;
        }
        return fetch(event.request)
          .catch(() => {
            // Optional: Return a fallback page or image when offline
            // return caches.match('/offline.html');
          });
      })
  );
});

// Background Sync event
self.addEventListener('sync', (event) => {
  if (event.tag === 'sync-tag-example') {
    console.log('[Service Worker] Background Sync triggered');
```

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

// Example background task
async function doSomeBackgroundTask() {
  console.log('Doing background task...');
  // Example: Sync pending requests to server
  // You can implement IndexedDB to store offline requests
}

// Push Notification event
self.addEventListener('push', (event) => {
  console.log('[Service Worker] Push Received.');
  let data = { title: 'Default Title', message: 'Default message' };

  if (event.data) {
    data = event.data.json();
  }

  const title = data.title;
  const options = {
    body: data.message,
    icon: '/icons/icon-192x192.png', // Updated with your cached icon
    badge: '/icons/icon-192x192.png', // Updated with your cached icon
  };

  event.waitUntil(
    self.registration.showNotification(title, options)
  );
});

// Optional: Handle notification click event
self.addEventListener('notificationclick', (event) => {
  event.notification.close();
  event.waitUntil(
    clients.openWindow('/') // Open homepage or any specific page
  );
});
```

**Output:**




```
// Background sync registration (simulate saving vocab progress)
if ('serviceWorker' in navigator && 'SyncManager' in window) {
    navigator.serviceWorker.ready.then(registration => {
      return registration.sync.register('sync-vocab-progress');
    }).then(() => {
      console.log("Background Sync Registered ✅");
    }).catch(console.error);
  }

  // Ask for push notification permission
  Notification.requestPermission().then(permission => {
    if (permission === "granted") {
      console.log("🔔 Notification permission granted.");
    }
  });
```

**Conclusion:**

By coding and registering a service worker, we enabled offline support and caching capabilities for the E-commerce PWA. The successful installation and activation of the service worker enhanced the app's reliability and performance, especially in low or no internet connectivity.

# 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”.

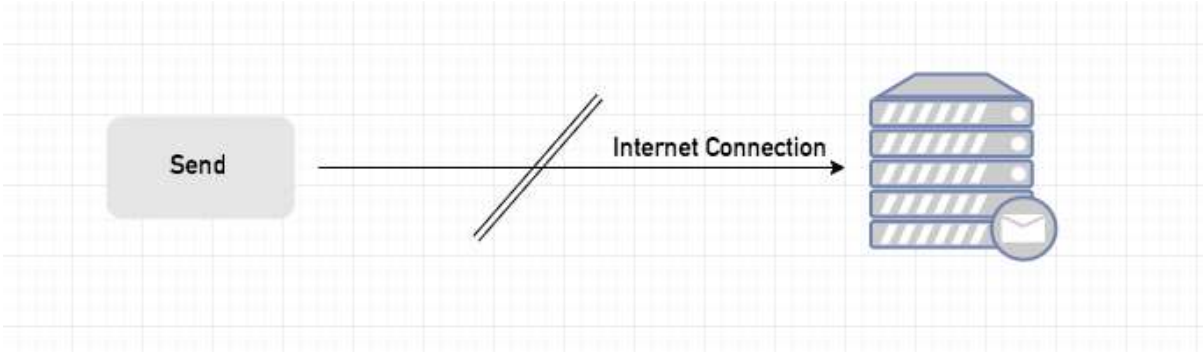
- **CacheFirst** - In this function, if the received request has cached before, the cached response is returned to the page. But if not, a new response requested from the network.
- **NetworkFirst** - In this function, firstly we can try getting an updated response from the network, if this process completed 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.

**Sync Event**

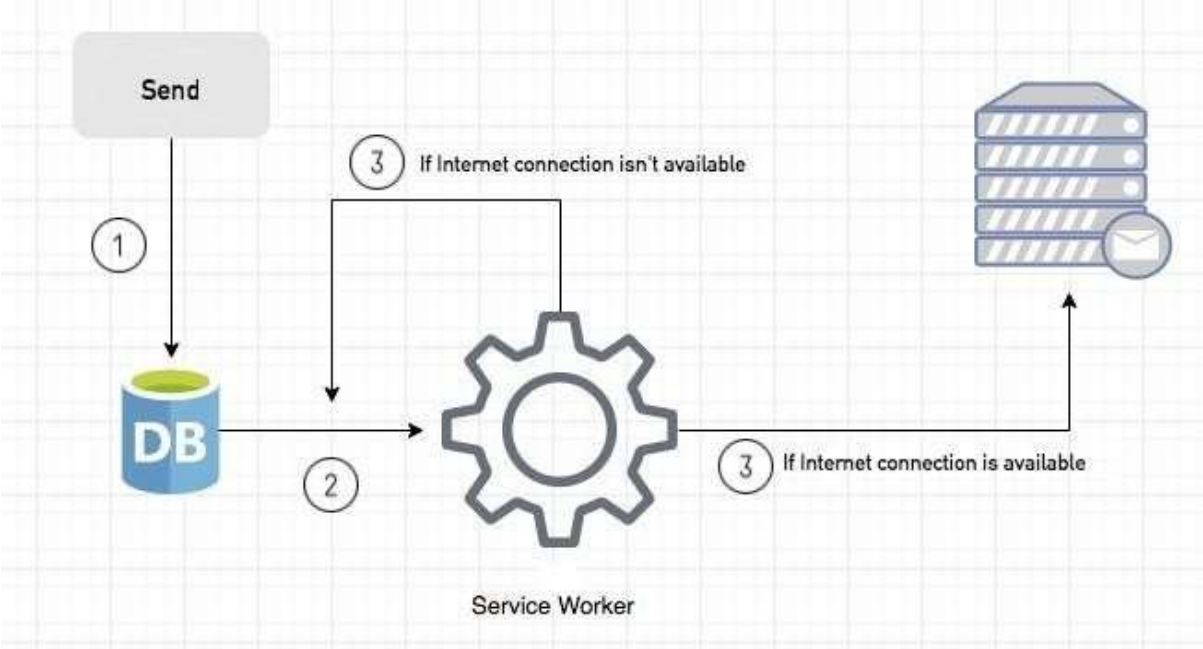
Background Sync is a Web API that is used to delay a process until the Internet connection is stable. We can adapt this definition to the real world; there is an e-mail client application that works on the browser and we want to send an email with this tool. Internet connection is broken while we are 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

```
document.querySelector("button").addEventListener("click", async () => {
    var swRegistration = await navigator.serviceWorker.register("sw.js");
    swRegistration.sync.register("helloSync").then(function () {
        console.log("helloSync success [main.js]");
    });
});
```

Event Listener for sw.js

```
self.addEventListener('sync', event => {
    if (event.tag == 'helloSync') {
        console.log("helloSync [sw.js]");
    }
});
```

**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.

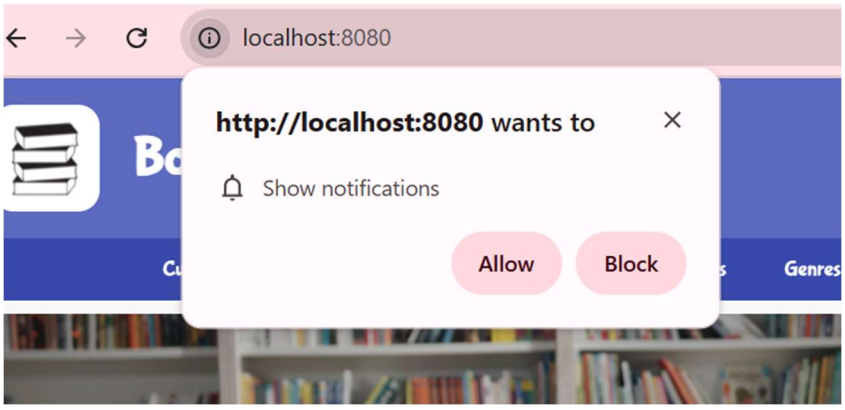
In the following code block is in sw.js file. You can handle push notifications with this event. In this example, I kept it simple. We send an object that has “method” and “message” properties. If the method value is “pushMessage”, we open the information notification with the “message” proper

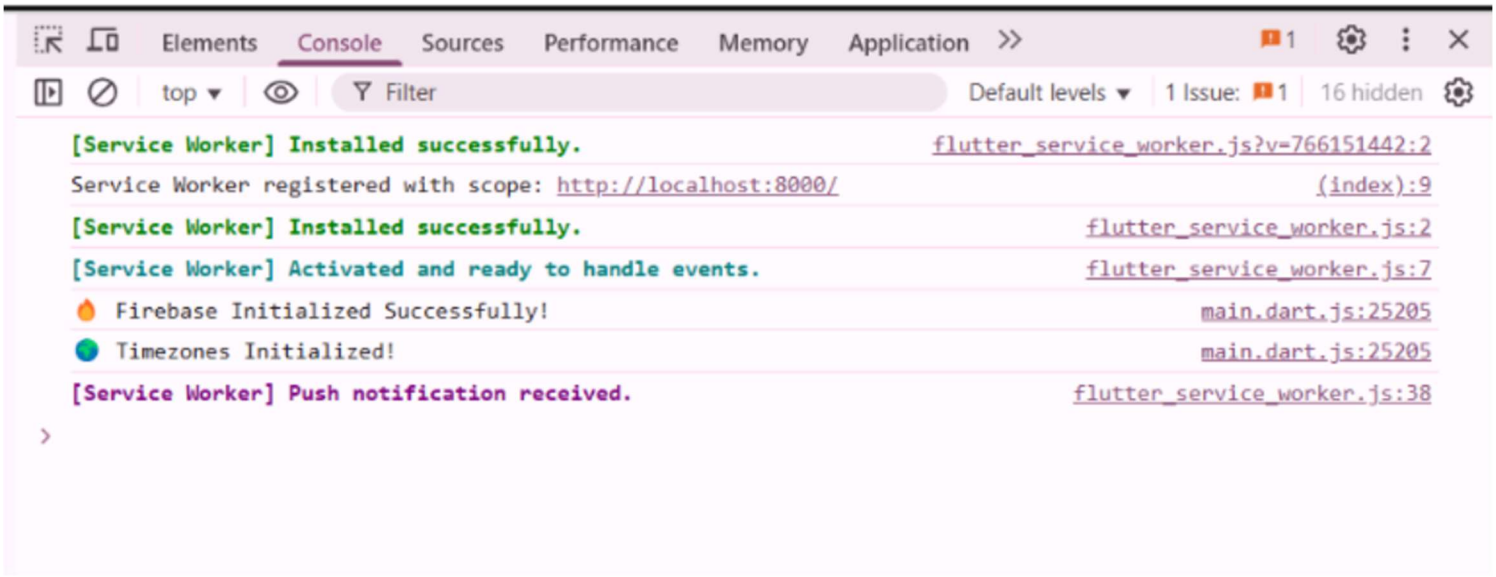
Screenshots:

```
// Push Notification event
self.addEventListener('push', (event) => {
  console.log('[Service Worker] Push Received.');
  let data = { title: 'Default Title', message: 'Default message' };

  if (event.data) {
    data = event.data.json();
  }

  const title = data.title;
  const options = {
    body: data.message,
    icon: '/icons/icon-192x192.png', // Updated with your cached icon
    badge: '/icons/icon-192x192.png', // Updated with your cached icon
  };
```

```
// Fetch event: serve from cache first, then network fallback
self.addEventListener('fetch', (event) => {
  event.respondWith(
    caches.match(event.request)
      .then((cachedResponse) => {
        if (cachedResponse) {
          return cachedResponse;
        }
        return fetch(event.request)
          .catch(() => {
            // Optional: Return a fallback page or image when offline
            // return caches.match('/offline.html');
          });
      })
  );
});
```

```
// Background Sync event
self.addEventListener('sync', (event) => {
  if (event.tag === 'sync-tag-example') {
    console.log('[Service Worker] Background Sync triggered');
    event.waitUntil(doSomeBackgroundTask());
  }
});
```




**Conclusion**:
By implementing fetch, sync, and push events in the service worker, we improved the performance, offline support, and user engagement of the PWA. These features enable real-time updates, background tasks, and seamless user experiences even with limited connectivity.

# Experiment No. 10

## Aim:

To study and implement deployment of PWA to GitHub Pages.

## Theory:

### GitHub Pages

Public web pages are freely hosted and easily published. Public webpages hosted directly from your GitHub repository. Just edit, push, and your changes are live.

GitHub Pages provides the following key features:

1. Blogging with Jekyll
2. Custom URL
3. Automatic Page Generator

Reasons for favoring this over Firebase:

1. Free to use
2. Right out of github
3. Quick to set up

GitHub Pages is used by Lyft, CircleCI, and HubSpot.
GitHub Pages is listed in 775 company stacks and 4401 developer stacks.

Pros

1. Very familiar interface if you are already using GitHub for your projects.
2. Easy to set up. Just push your static website to the gh-pages branch and your website is ready.
3. Supports Jekyll out of the box.
4. Supports custom domains. Just add a file called CNAME to the root of your site, add an A record in the site's DNS configuration, and you are done.

Cons

1. The code of your website will be public, unless you pay for a private repository.
2. Currently, there is no support for HTTPS for custom domains. It's probably coming soon though.
3. Although Jekyll is supported, plug-in support is rather spotty.

## Firebase

The Realtime App Platform. Firebase is a cloud service designed to power real-time, collaborative applications. Simply add the Firebase library to your application to gain access to a shared data structure; any changes you make to that data are automatically synchronized with the Firebase cloud and with other clients within milliseconds.

Some of the features offered by Firebase are:

1. Add the Firebase library to your app and get access to a shared data structure. Any changes made to that data are automatically synchronized with the Firebase cloud and with other clients within milliseconds.
2. Firebase apps can be written entirely with client-side code, update in real-time out-of-the-box, interoperate well with existing services, scale automatically, and provide strong data security.
3. Data Accessibility- Data is stored as JSON in Firebase. Every piece of data has its own URL which can be used in Firebase's client libraries and as a REST endpoint. These URLs can also be entered into a browser to view the data and watch it update in real-time.

Reasons for favoring over GitHub Pages:

1. Realtime backend made easy
2. Fast and responsive

Instacart, 9GAG, and Twitch are some of the popular companies that use Firebase
Firebase has a broader approval, being mentioned in 1215 company stacks & 4651 developers stacks

Pros

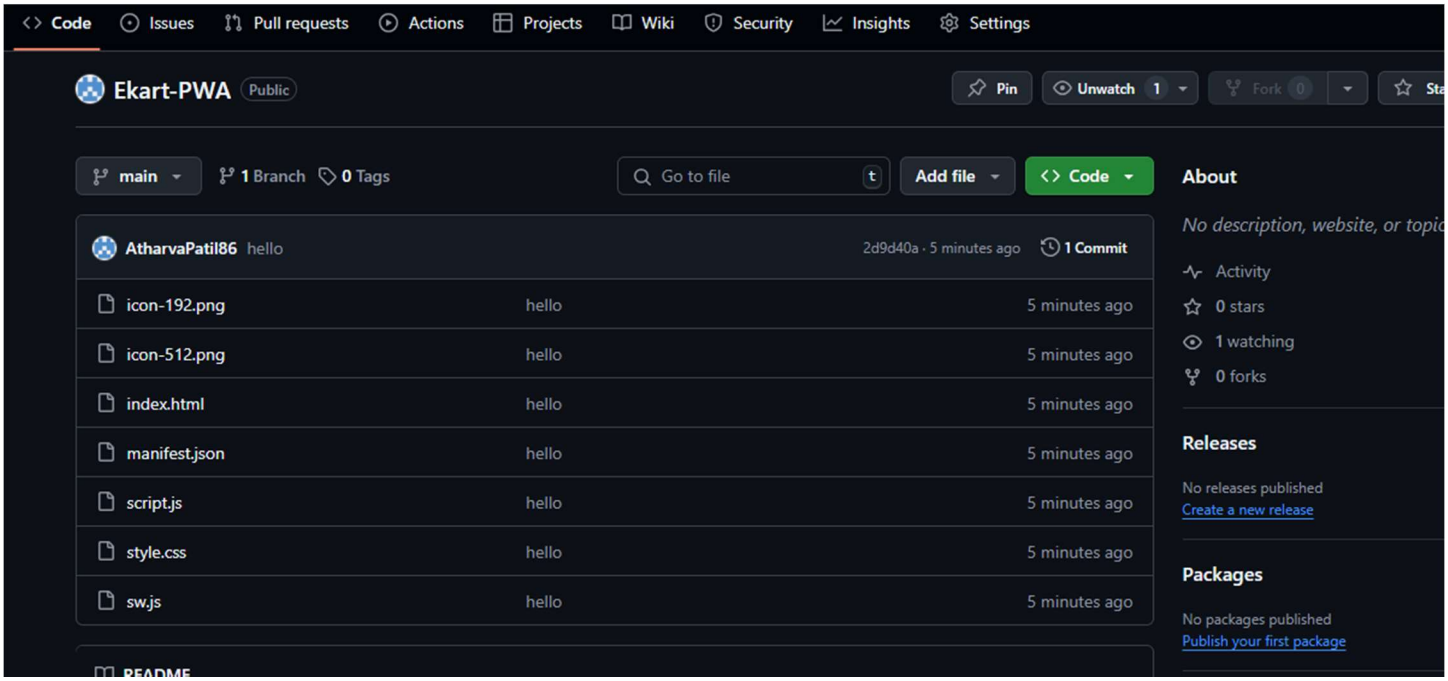
1. Hosted by Google. Enough said.
2. Authentication, Cloud Messaging, and a whole lot of other handy services will be available to you.
3. A real-time database will be available to you, which can store 1 GB of data.
4. You'll also have access to a blob store, which can store another 1 GB of data.
5. Support for HTTPS. A free certificate will be provisioned for your custom domain within 24 hours.

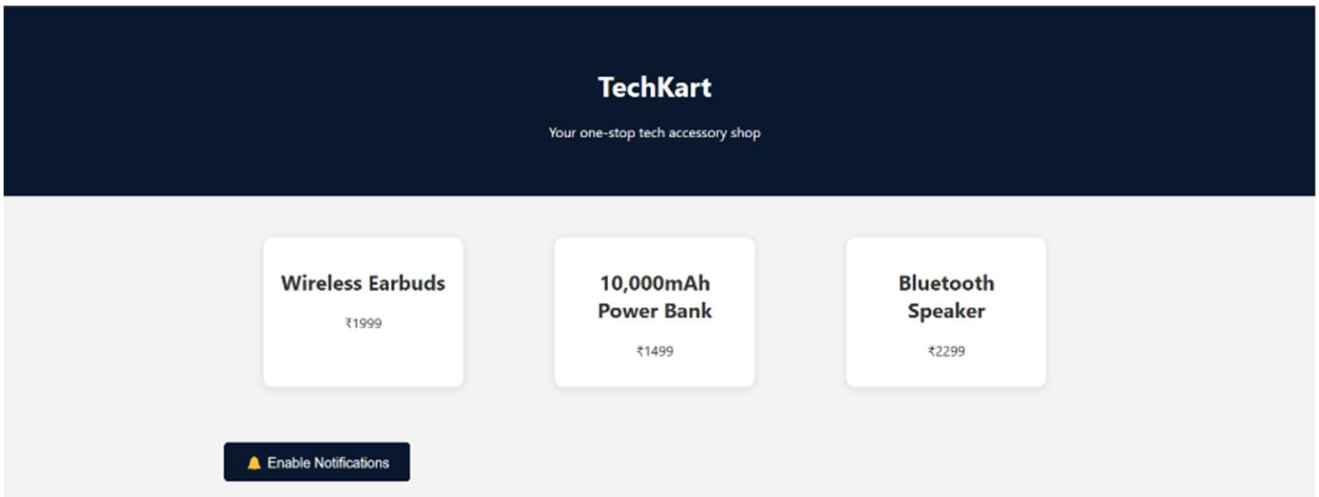
Cons

1. Only 10 GB of data transfer is allowed per month. But this is not really a big problem, if you use a CDN or AMP.
2. Command-line interface only.
3. No in-built support for any static site generator.

**Link to GitHub repository:** https://github.com/AtharvaPatil86/Ekart-PWA

## Github Screenshot:



**Conclusion**:
By successfully deploying the PWA to GitHub Pages, we made the app accessible online through a live URL. This experiment helped us understand the deployment process and the importance of version control and static hosting for PWAs.

# Experiment No. 11

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

**Theory**:

**Google Lighthouse:**

Google Lighthouse is a tool that lets you audit your web application based on a number of parameters including (but not limited to) performance, based on a number of metrics, mobile compatibility, Progressive Web App (PWA) implementations, etc. All you have to do is run it on a page or pass it a URL, sit back for a couple of minutes and get a very elaborate report, not much short of one that a professional auditor would have compiled in about a week.

The best part is that you have to set up almost nothing to get started. Let's begin by looking at some of the top features and audit criteria used by Lighthouse.

**Key Features and Audit Metrics**

Google Lighthouse has the option of running the Audit for Desktop as well as mobile version of your page(s). The top metrics that will be measured in the Audit are:

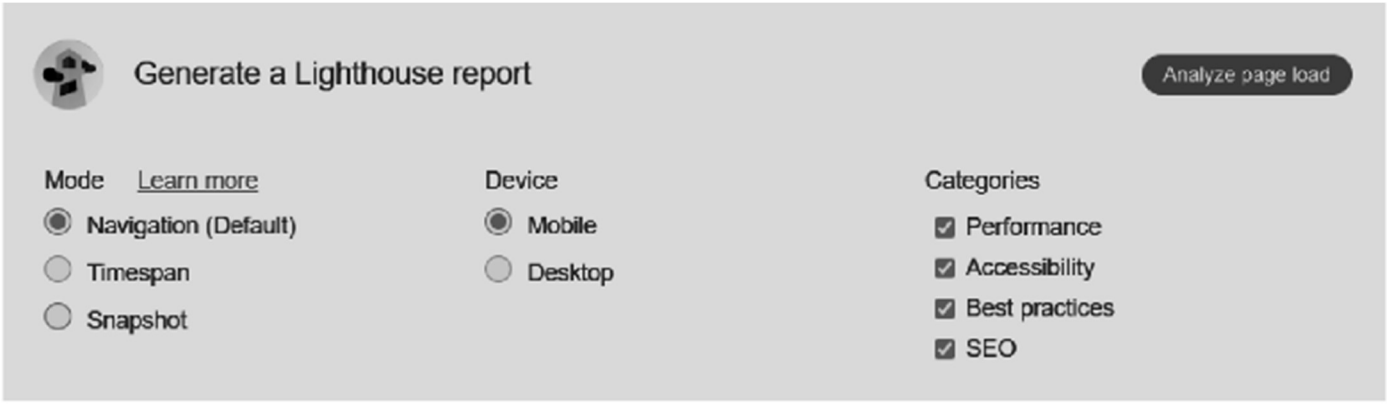
1. **Performance:** This score is an aggregation of how the page fared in aspects such as (but not limited to) loading speed, time taken for loading for basic frame(s), displaying meaningful content to the user, etc. To a layman, this score is indicative of how decently the site performs, with a score of 100 meaning that you figure in the 98th percentile, 50 meaning that you figure in the 75th percentile and so on.
2. **PWA Score (Mobile):** Thanks to the rise of Service Workers, app manifests, etc., a lot of modern web applications are moving towards the PWA paradigm, where the objective is to make the application behave as close as possible to native mobile applications. Scoring points are based on the Baseline PWA checklist laid down by Google which includes Service Worker implementation(s), viewport handling, offline functionality, performance in script-disabled environments, etc.
3. **Accessibility:** As you might have guessed, this metric is a measure of how accessible your website is, across a plethora of accessibility features that can be implemented in your page (such as the 'aria-' attributes like aria-required, audio captions, button names,

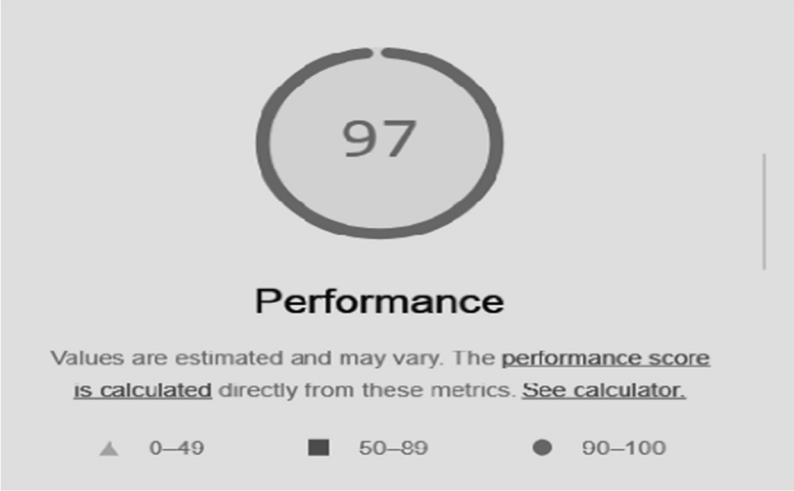
etc.). Unlike the other metrics though, Accessibility metrics score on a pass/fail basis

i.e. if all possible elements of the page are not screen-reader friendly (HTML5 introduced features that would make pages easy to interpret for screen readers used by visually challenged people like tag names, tags such as <section>, <article>, etc.), you get a 0 on that score. The aggregate of these scores is your Accessibility metric score.

4. **Best Practices:** As any developer would know, there are a number of practices that have been deemed 'best' based on empirical data. This metric is an aggregation of many such points, including but not limited to:Use of HTTPS
   Avoiding the use of deprecated code elements like tags, directives, libraries, etc.
   Password input with paste-into disabled
   Geo-Location and cookie usage alerts on load, etc.

**Screenshots:**




The performance of a Progressive Web App (PWA) was evaluated using the **Google Lighthouse** tool. Lighthouse is an open-source, automated tool used to improve the quality of web pages. It audits performance, accessibility, best practices, and SEO, providing developers with detailed feedback and actionable suggestions.

The PWA was tested using Lighthouse, and the results showed excellent scores in **Performance (96)**, **Accessibility (100)**, and **Best Practices (93)**, while the **SEO score (82)** indicated areas that could benefit from improvement. These metrics help assess how well the application adheres to modern web development standards, offering insights into both the strengths and potential areas of optimization.

Github Link: https://github.com/AtharvaPatil86/Ekart-PWA

**Conclusion:** In this experiment, the Google Lighthouse PWA Analysis Tool was used to evaluate the functioning of a Progressive Web App. The tool provided comprehensive insights into SEO, accessibility, and best practices. It helped identify areas for improvement and ensured the application adheres to modern web standards, resulting in a more optimized, user-friendly, and reliable web experience.