Aim: Installation and Configuration of Flutter Environment.

Theory:

Flutter is an open-source UI toolkit developed by Google that enables the development of applications for multiple platforms including Android, iOS, web, desktop, and embedded systems using a single codebase. It uses the Dart programming language and offers a rich set of pre-designed widgets for crafting beautiful and natively compiled applications.

Key Features of Flutter:

● Single Codebase: Write once and run on multiple platforms.

● Hot Reload: Instantly see changes during development.

● Custom Widgets: Access to a wide collection of widgets and customization options.

● High Performance: Compiles to ARM or x86 native libraries.

● Open Source: Free to use and backed by an active community.

Requirements:

For Windows:

● Windows 10 or later (64-bit)

● 1.64 GB of disk space (excluding IDE/tools)

● Git for Windows

For macOS:

● macOS (Intel or Apple Silicon)
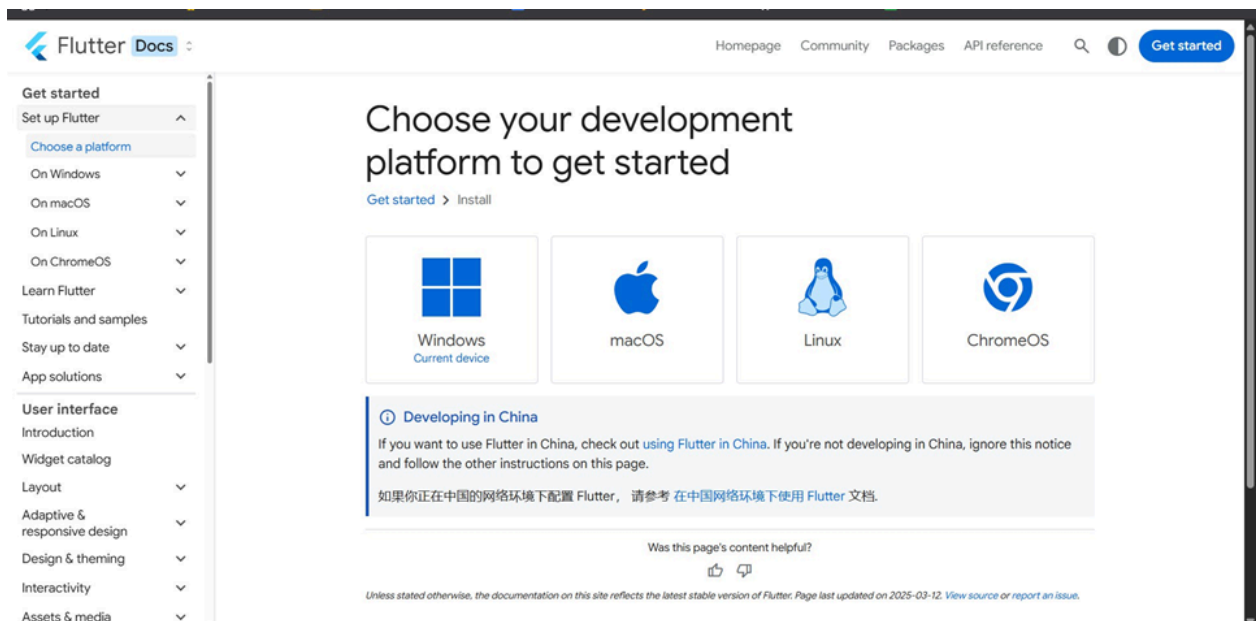
● Xcode for iOS development

For Linux:

● 　Any recent Linux distribution with bash, curl, git, and unzip

Installation Steps:

Step 1: Download the Flutter SDK

● 　Navigate to the official Flutter website: https://flutter.dev

● 　Select your operating system and download the latest stable release.

● 　Extract the contents to a desired location (e.g., C:\flutter on Windows or

~/development/flutter on Linux/macOS).

Step 2: Set Up System Environment Variable

● 　Add the Flutter SDK's bin directory to your system PATH.

○ 　On Windows: Add C:\flutter\bin to Environment Variables.

○ 　On macOS/Linux: Add the line export PATH="$PATH:[PATH_TO_FLUTTER_DIRECTORY]/bin" to your shell configuration file (e.g., .bashrc, .zshrc).

Step 3: Verify Installation

Open the terminal or command prompt and run:

flutter doctor

This command checks your environment and displays a report of the Flutter installation along with required dependencies.

Step 4: Install a Code Editor or IDE You can use:

●   Android Studio (recommended for full-featured Flutter development)

●   Visual Studio Code (lightweight and fast)

Step 5: Install Flutter and Dart Plugins In Android Studio:

●   Open Settings > Plugins

●   Search and install the Flutter plugin

●   The Dart plugin will be prompted for installation automatically

In Visual Studio Code:

●   Open the Extensions tab

●   Search for and install both "Flutter" and "Dart"
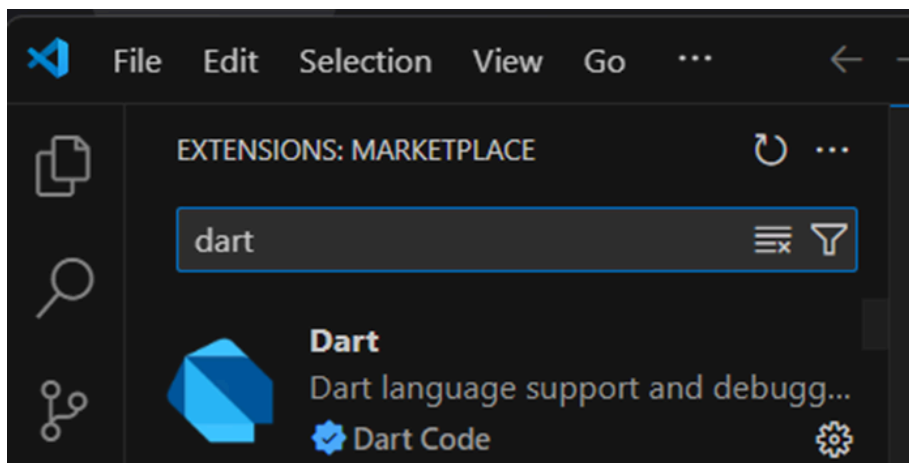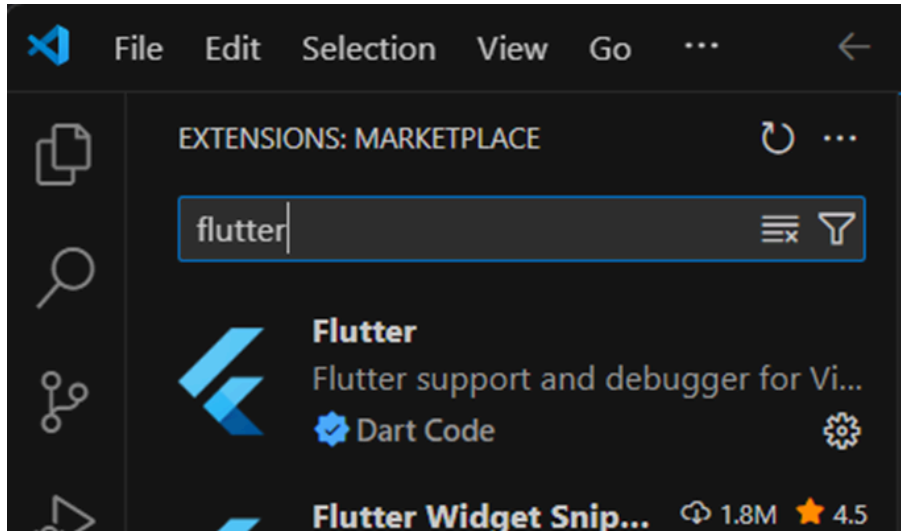
Step 6: Set Up an Android Emulator (Optional for Testing)

● Open Android Studio

● Go to AVD Manager and create a new virtual device

● Choose a hardware profile and system image

● Launch the emulator

Step 7: Create and Run a Sample Application

Use the following commands to create a basic Flutter project and run it: flutter create myapp

cd myapp flutter run

Make sure a simulator or physical device is connected and running.

Conclusion: The Flutter environment has been successfully installed and configured. Developers can now build and test cross-platform applications using a single codebase. The setup ensures a consistent development experience with native performance and modern UI capabilities.

Aim: To design Flutter UI by including common widgets.

Theory:

Flutter is a modern UI toolkit developed by Google that allows developers to build natively compiled applications for mobile, web, and desktop platforms from a single codebase. It provides a rich set of pre-built widgets and tools that make UI development fast, flexible, and expressive. One of the most fundamental principles in Flutter is that everything is a widget—from layout components to user interface elements and even invisible helpers like padding.

Widgets in Flutter are used to build the entire UI by composing smaller widgets into complex hierarchies. These widgets fall into different categories based on their role in the user interface.

Types of Flutter Widgets:

Widgets in Flutter can be broadly categorized into five main types:

1.   **Structural Widgets**

Structural widgets are responsible for defining the basic structure of a Flutter application. These are often the first widgets used when building a screen and include top-level components that provide visual scaffolding and layout organization.

Examples:

●   MaterialApp: Wraps the entire app and applies Material Design throughout.

●   Scaffold: Provides a framework for implementing standard app elements like the AppBar, body, floating action buttons, drawers, etc.

●   AppBar: Represents a material design app bar, often used as the top navigation bar of the screen.

●   Container: A versatile widget used for styling and positioning its child widgets with padding, margins, borders, background color, etc.

These widgets form the foundation upon which the UI is built.

2.   **Layout Widgets**

Layout widgets are used to arrange and position other widgets within the user interface. They control how widgets are nested, spaced, aligned, and layered.

Examples:

●   Column: Arranges child widgets vertically.

●   Row: Arranges child widgets horizontally.

●   Stack: Overlays widgets on top of each other (like layers), allowing complex UI designs.

●   Padding: Adds space around a widget.

●   SizedBox: Adds fixed space between widgets or defines a fixed width or height for a widget.

Layout widgets allow developers to organize UI elements precisely as needed, offering control over the visual composition.

3.   **Display Widgets**

Display widgets are used to present static content such as text, images, or icons on the screen. These widgets are essential for conveying information to the user.

Examples:

●   Text: Displays a string of text with customizable styles.

●   Image: Displays an image from an asset, file, or network.

●   Icon: Renders predefined icons from material or custom icon sets.

These widgets are typically non-interactive and focus purely on the visual representation of data.

4.   **Scrollable Widgets**

Scrollable widgets are essential when the content exceeds the screen size and needs to be viewed through scrolling. They make the UI responsive and user-friendly on devices with limited screen space.

Examples:

● ListView: Displays a scrollable list of widgets. It can be built dynamically using builders for large datasets.

● SingleChildScrollView: Makes a single widgetscrollable when content overflows vertically or horizontally.

● GridView: Displays content in a 2D scrollable grid layout.

These widgets are crucial for building scrollable views and dynamic lists in real-world applications.

5.   **Interactive Widgets**

Interactive widgets are responsible for handling user input such as taps, gestures, and other actions. They enhance user engagement by providing interactive elements like buttons and feedback mechanisms.
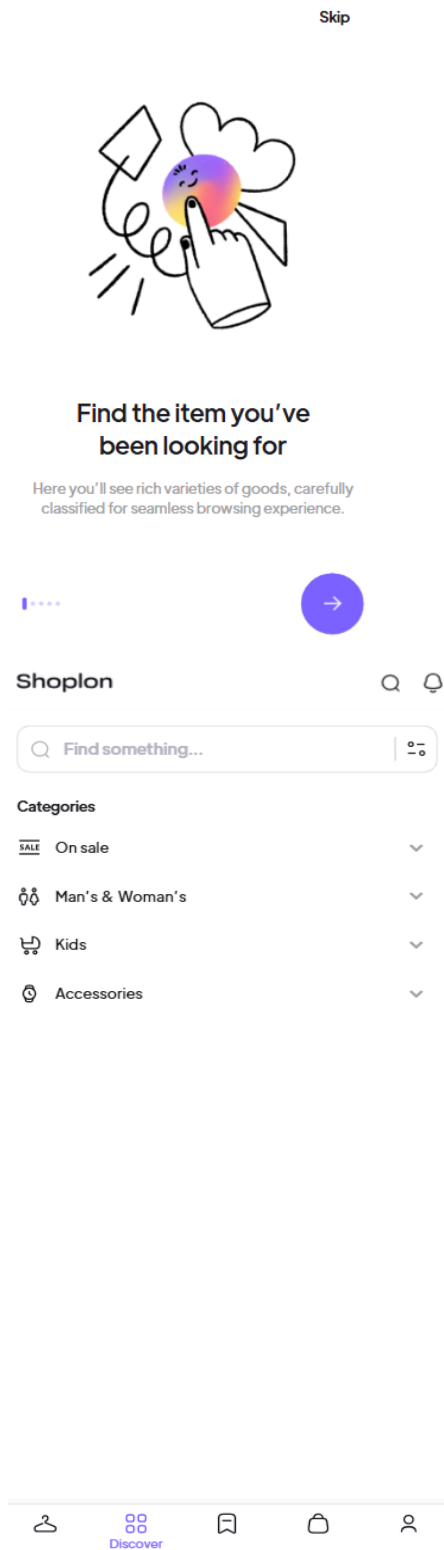
Examples:

● GestureDetector: Detects gestures like taps, double taps, long presses, etc., and triggers actions accordingly.

● ElevatedButton: A material design button with elevation and built-in onPress functionality.

● TextButton: A simple button without elevation or background.

● SnackBar: Provides brief messages at the bottom of the screen as a form of feedback or notification.

These widgets enable developers to capture user interactions and respond with appropriate UI behavior or actions.

Code (Github): https://github.com/Kingmaker-2/MPL-APP.git

Output:

Skip

Find the item you've
been looking for

Here you'll see rich varieties of goods, carefully
classified for seamless browsing experience.

Shoplon

Find something...

Categories

SALE　On sale

Man's & Woman's

Kids

Accessories

Discover

Conclusion: Understanding the different types of widgets in Flutter is essential for designing effective and maintainable user interfaces. Each widget type serves a specific role—whether it's structuring the layout, displaying content, enabling interactivity, or handling overflow through scrolling. By combining these widgets effectively, developers can create flexible and scalable UIs tailored to a wide variety of applications.In the provided experiment, all these widget types are demonstrated within a simple Flutter application, showcasing their usage in a real-world scenario. This hands-on practice reinforces the concept of widget-based UI design and prepares developers to build more complex applications using Flutter's powerful toolkit.

Aim: To include icons, fonts in Flutter app

Theory:

Flutter is an open-source UI toolkit developed by Google for building beautiful, natively compiled applications for mobile, web, and desktop from a single codebase. In this experiment, we explore three key UI elements:

- **Icons** – used to represent actions or content visually.

- **Images** – used to enhance visual appeal and deliver information graphically.

- **Fonts** – used to style the text to match branding and improve readability.

Each of these components plays a vital role in enhancing user interface and user experience.

Icons in Flutter:

Icons are visual representations of commands, files, directories, devices, or common actions. They help users quickly understand functionality without relying on text labels.

Flutter provides a rich set of Material Design Icons out of the box through the Icons class. These are scalable vector icons which can be resized, colored, or styled as needed.

Types of Icons:

- Material Icons: Predefined and bundled with Flutter.

- Custom Icon Fonts: Imported manually or using external packages like font_awesome_flutter.

Basic Syntax for Using Icons: Icon(Icons.icon_name) Example:

Icon(Icons.home)

With styling:

Icon(Icons.person, color: Colors.teal, size: 30)

Fonts in Flutter

Fonts determine the style and appearance of textual content in the app. Using custom fonts in your Flutter app gives you more control over branding and presentation.

Default Font:

- Flutter uses Roboto as its default font.

Custom Fonts:

You can import and use any .ttf or .otf font in your Flutter project by following these steps:

Step 1: Add font files Place your font files (e.g., Poppins-Regular.ttf) in the directory: assets/fonts/

Step 2: Declare in pubspec.yaml

fonts:

- family: Poppins fonts:

- asset: assets/fonts/Poppins-Regular.ttf

Step 3: Use the font in your Flutter app Apply the font globally in ThemeData: theme: ThemeData(

fontFamily: 'Poppins',

)

Or apply it locally in a Text widget:

Text("Hello", style: TextStyle(fontFamily: 'Poppins'))

Images in Flutter

Images are used to visually convey information, decorate the app, or present media. Flutter allows you to use images from assets, files, or the network.

Types of Images:

- Asset Images: Stored in the app's asset directory.

- Network Images: Loaded from a web URL.

- File Images: Loaded from the device storage.

How to Use Asset Images:

Step 1: Place the image in your assets folder

Example path: assets/images/sample.jpg

Step 2: Declare the image in pubspec.yaml

assets:

- assets/images/sample.jpg

Step 3: Use it in your code

Image.asset("assets/images/sample.jpg")

For styled use with decorations:

Container(height: 180,

width: double.infinity, decoration: BoxDecoration(

borderRadius: BorderRadius.circular(12), image: DecorationImage(

image: AssetImage("assets/images/sample.jpg"), fit: BoxFit.cover,

),

),

)

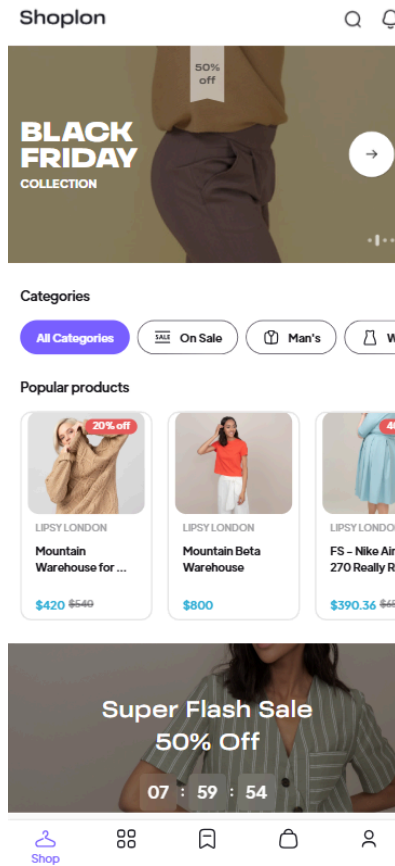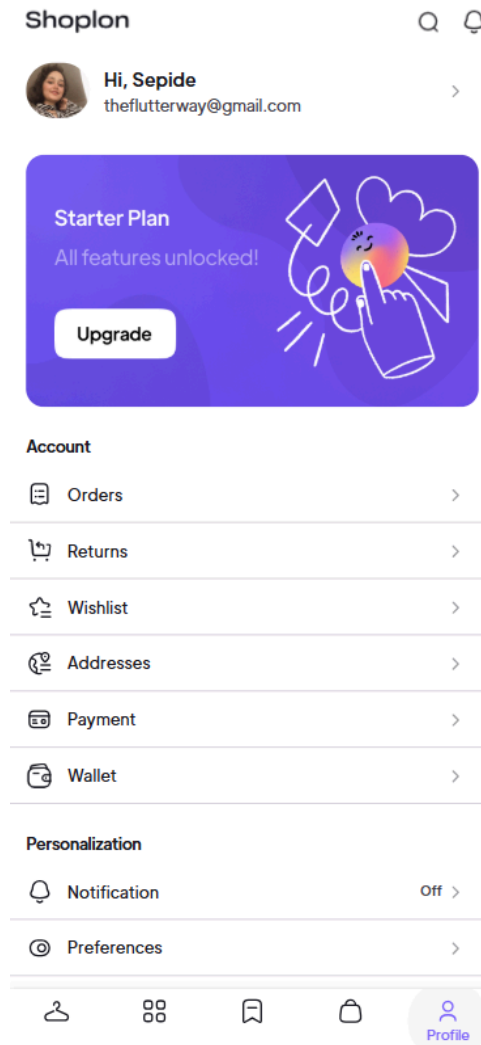**Code (Github):** https://github.com/Kingmaker-2/MPL-APP.git

Output:

Conclusion: In conclusion, this experiment provided a comprehensive understanding of how to integrate icons, images, and custom fonts in a Flutter application. Icons enhance user interaction by visually representing actions and navigation elements, making the interface more intuitive. Custom fonts allow developers to personalize the appearance of text, aligning the design with branding requirements and improving readability. Images, on the other hand, add visual appeal and context, contributing to a richer user experience. By combining these elements effectively, developers can create visually engaging and user-friendly mobile applications. This foundational knowledge is essential for building polished, professional Flutter apps.

Aim: To create an interactive Form using form widget

Theory:

In mobile app development, forms are essential for collecting user input, such as login details, preferences, or order information. Flutter provides a robust way to create forms that can manage validation, saving input, and handling submission events.

A form is a container that groups together multiple input fields (like text fields, dropdowns, checkboxes) and allows collective validation and saving of user input. It ensures that the input meets certain criteria before processing or submitting it.

Form Widget in Flutter

The Form widget in Flutter acts as a container for grouping and managing multiple input widgets like TextFormField, DropdownButtonFormField, etc. It uses a GlobalKey<FormState> to uniquely identify the form and enable form-wide operations such as:

● Validation: Checking if all inputs satisfy validation rules.

● Saving: Triggering callbacks to save the current input values.

● Resetting: Clearing or resetting the form fields.

By using a Form widget, developers can efficiently handle complex user input scenarios in a clean and maintainable way.

Basic Syntax of Forms in Flutter:

The typical steps to implement a form in Flutter include:

Define a GlobalKey for the FormState:

final _formKey = GlobalKey<FormState>();

Wrap input widgets inside a Form widget:

Form(

key: _formKey, child: Column(children: [

```
TextFormField( validator: (value) {

if (value == null || value.isEmpty) { return 'Please enter some text';

}

return null;

},

onSaved: (value) {

// Save the value to a variable

},

),

ElevatedButton( onPressed: () {

if (_formKey.currentState!.validate()) {

_formKey.currentState!.save();

// Process the data

}

},

child: Text('Submit'),

),

],

),

);
```

1.   **Validation and Saving:**

○   Each input widget like TextFormField has a validator function that returns a validation error string or null if valid.

○   The onSaved callback captures and stores the input value when the form is saved.

2.   **Submit Handling:**

○   On submit, call _formKey.currentState!.validate() to validate all fields.

○   If valid, call _formKey.currentState!.save() to invoke onSaved for all fields.

Code: https://github.com/Kingmaker-2/MPL-APP.git

Output:

Basic form created

Conclusion: By creating an interactive form using the Form widget in Flutter, we have learned how to efficiently collect, validate, and manage user input within a mobile application. Understanding form fields, validation logic, and state management enables you to build robust and user-friendly interfaces for features like registration, login, and data entry. This forms an essential part of developing interactive and responsive Flutter applications.

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

Theory:

Navigation in Flutter is the mechanism of transitioning between different screens or views (also called routes). Flutter follows a stack-based navigation model where screens are pushed and popped from the stack.

Purpose of Navigation:

● Allows movement between various UI components (screens)

● Helps build multi-screen mobile applications.

● Maintains user flow and logical screen transitions.

Basic Syntax:

● Navigate to a new screen:

Navigator.push(context, MaterialPageRoute(builder: (context) => NextScreen()));

● Navigate using a named route: Navigator.pushNamed(context, '/form');

● Go back to the previous screen: Navigator.pop(context);

Routing in Flutter:

Routing refers to the configuration of named paths and their corresponding screen widgets in an application. Flutter provides both named and unnamed routing options.

Importance of Routing:

● Organizes navigation paths in a centralized way.

● Makes large applications easier to manage and scale.

● Enables structured access to screens via route names.

Basic Syntax:

● Define routes in MaterialApp: MaterialApp(

initialRoute: '/', routes: {

'/': (context) => HomeScreen(), '/form': (context) => FormScreen(),

},

)

- Use route name to navigate: Navigator.pushNamed(context, '/form');

Gestures in Flutter:

Gestures refer to touch-based interactions such as taps, swipes, drags, and long presses. Flutter provides the GestureDetector widget to recognize and handle these interactions.

Purpose of Using Gestures:

- Makes the app more interactive.

- Captures and responds to user input.

- Enhances user experience by adding custom actions to touch events.

Basic Syntax:

- Detect a tap: GestureDetector(onTap: () {

// Code to execute on tap

},

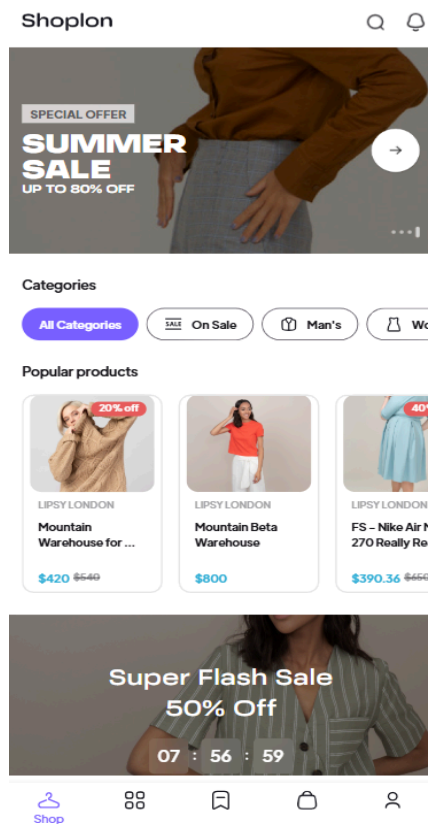child: Widget(), // Child widget to wrap

);

Other gestures include:

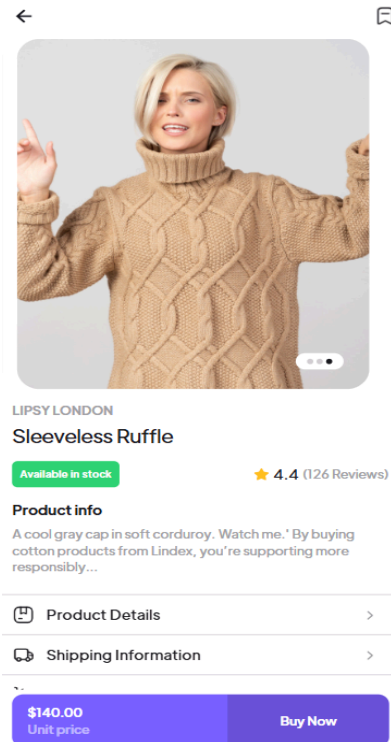- onDoubleTap

- onLongPress

- onHorizontalDragUpdate, etc.

Code: https://github.com/Kingmaker-2/MPL-APP.git
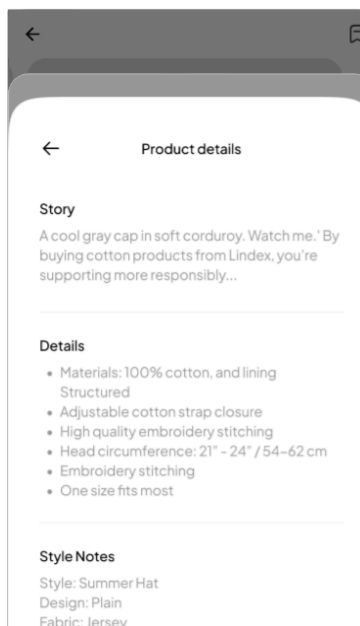
Output:

Page 1 (Home Page)

Page 2 Product Page (User clicked on one Product):

Product Details:

Add to Cart:
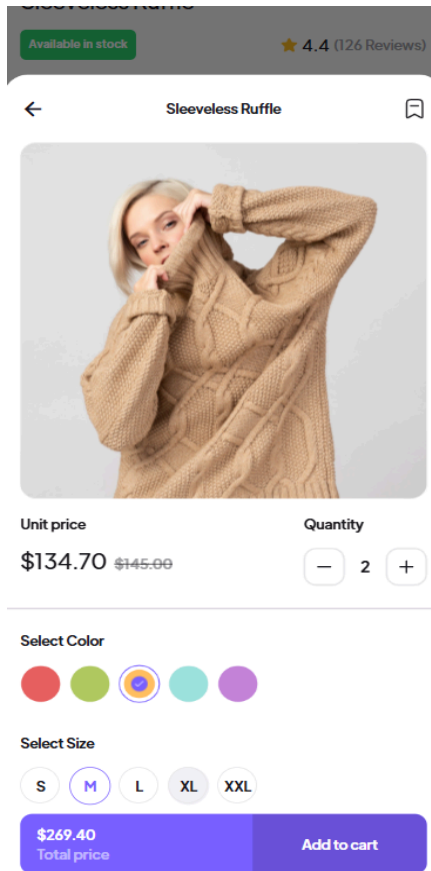
Conclusion: By applying navigation, routing, and gesture handling in a Flutter application, we have learned how to create smooth transitions between screens and build interactive user experiences. Understanding how to use Navigator, manage routes, and respond to gestures is essential for developing dynamic, multi-screen applications. These features play a key role in improving app usability and enabling intuitive user interactions.

Aim: To Connect Flutter UI with fireBase database

Theory:

What is Firebase?

Firebase is a platform developed by Google that offers Backend-as-a-Service (BaaS) to support the development of web and mobile applications. It provides a wide range of tools and services such as Firestore (a cloud-hosted NoSQL database), Realtime Database, Firebase Authentication, Cloud Storage, Cloud Functions, Analytics, Crash Reporting, and Push Notifications. These services allow developers to focus more on frontend development while Firebase handles backend tasks like data storage, user authentication, and serverless computing.

Why is Firebase Used?

Firebase is widely used in application development due to its real-time capabilities, scalability, and ease of integration. It allows developers to synchronize data across clients in real time without needing to manage their own servers. Firebase provides simple APIs and SDKs that reduce development time and complexity, especially for real-time applications like chat apps, collaborative tools, and live dashboards. Additionally, Firebase supports user authentication, secure data access, and automatic scaling, making it suitable for both small and large-scale applications.

Steps to Connect Firebase with Flutter UI

1.   **Create a Firebase Project**

Go to the Firebase Console (https://console.firebase.google.com/) and create a new project by providing a project name and enabling necessary services like Google Analytics if required.
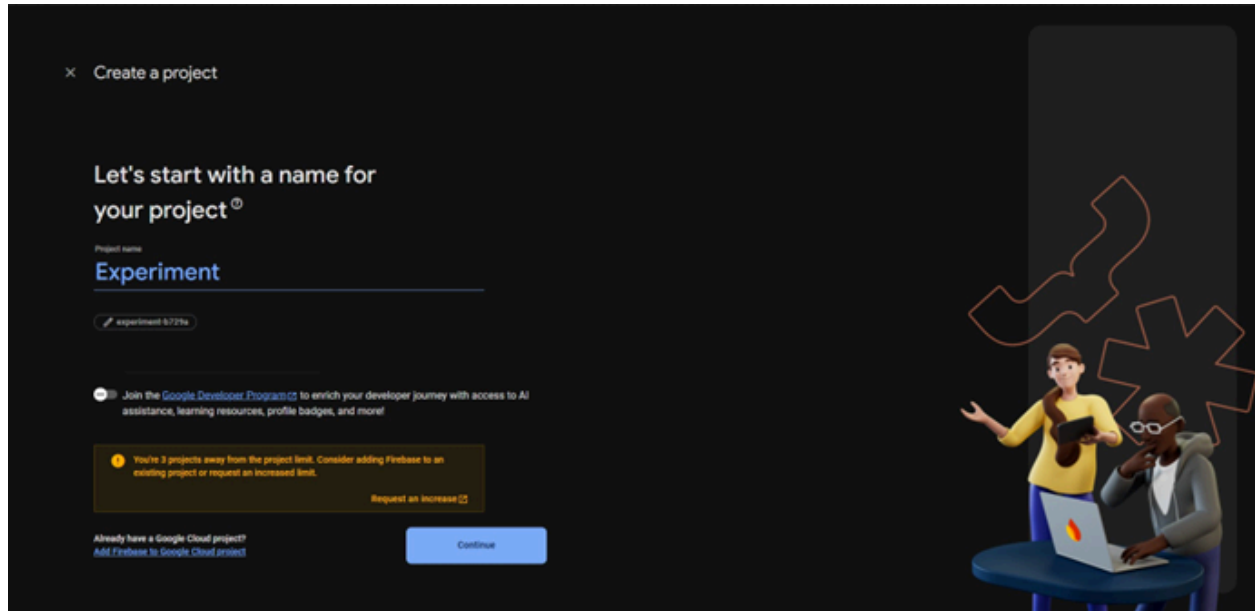
2.   Register Your App with Firebase

In the Firebase console, add a new application by selecting the Android or iOS platform. For Android, provide the package name, nickname, and SHA-1 key (if needed).
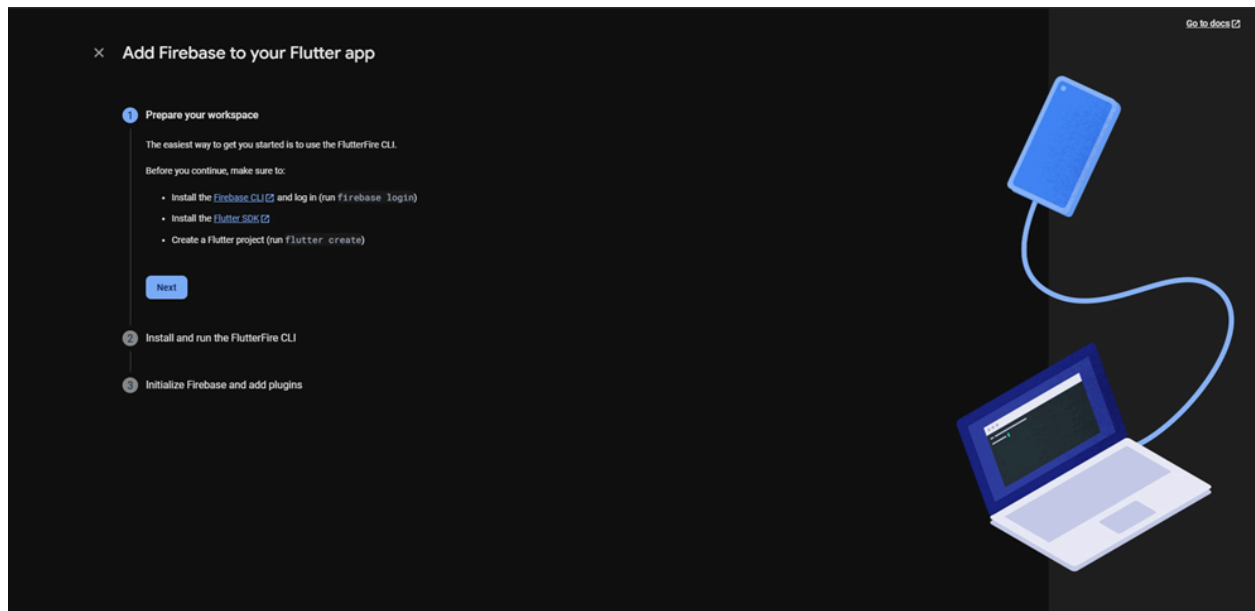
3.   Download and Add Configuration File

For Android, download the google-services.json file and place it in the android/app directory of your Flutter project. For iOS, download the GoogleService-Info.plist file and add it to the iOS runner project using Xcode.
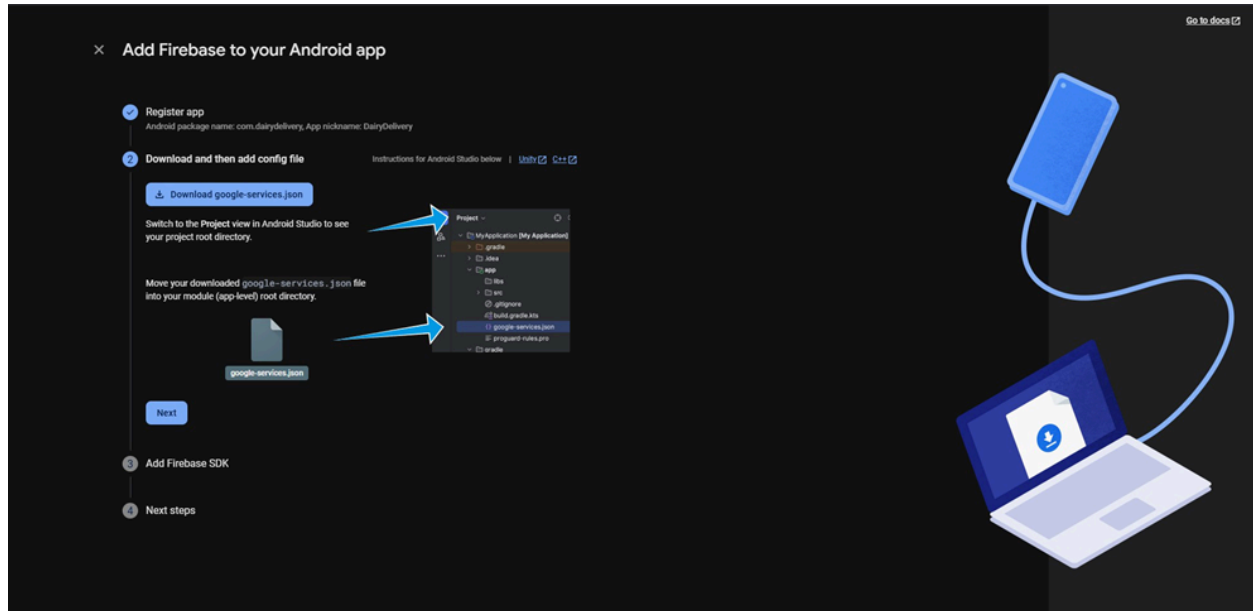
4.   **Add Firebase Dependencies**

Open your pubspec.yaml file and add necessary Firebase dependencies, such as:

dependencies:

firebase_core: latest_version

cloud_firestore: latest_version

 firebase_auth: latest_version

Then run the flutter pub get to fetch the packages.

5.   **Configure Firebase in Your App**

Import and initialize Firebase in your Flutter app. In the main.dart file, ensure Firebase is initialized before the app runs:

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

}

6.   **Use Firebase Services in the UI**

After initializing Firebase, you can use its services within your UI. For example, you can use FirebaseAuth for authentication and FirebaseFirestore to fetch or write data in the app interface.

7.   **Update Android and iOS Configuration**

For Android, make sure to apply the Google Services plugin in android/build.gradle and android/app/build.gradle. For iOS, ensure the platform version is at least 10.0 and required permissions are configured in Info.plist.

Output:
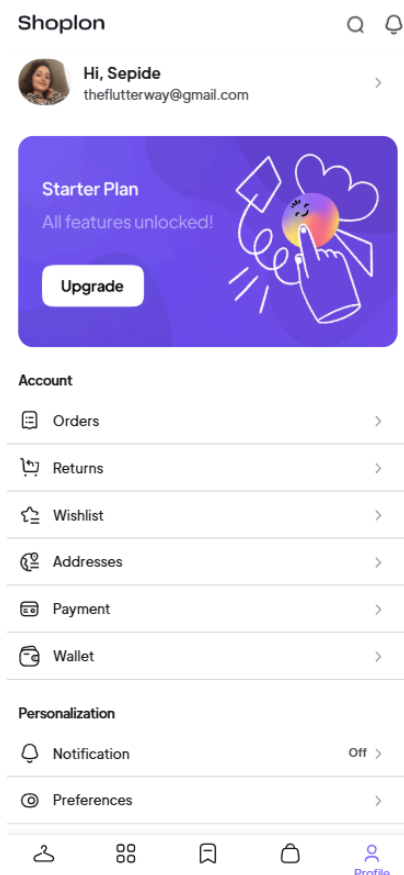
Conclusion: By successfully connecting the Flutter UI with the Firebase database, you have learned how to create dynamic and data-driven applications. This integration allows real-time data synchronization between the app and the backend, enabling

functionalities such as data storage, retrieval, and updates. Understanding how to connect Flutter with Firebase is essential for developing modern applications that require persistent data, user management, and seamless cloud connectivity.

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

Theory:
Introduction to Progressive Web Apps (PWAs):

A Progressive Web App (PWA) is a modern web application that offers an experience similar to native mobile applications. PWAs combine the best features of the web and mobile apps, including offline access, responsive design, fast loading times, and installation capabilities. One key feature of PWAs is the "Add to Homescreen" functionality, which allows users to install the web app directly onto their device's home screen, launching it in standalone mode like a native app.

Importance of the Web App Manifest:

To enable the "Add to Homescreen" feature, a PWA must include a Web App Manifest file, typically named manifest.json. This JSON file provides essential metadata about the application. It includes information such as the application's name, short name, icons, start URL, display mode, background color, and theme color. When a browser detects the manifest file and a registered service worker, it considers the web app installable and prompts the user with an option to add it to their home screen.

Structure and Role of the Manifest File:

The manifest file plays a critical role in defining how the app appears when installed. The name and short_name fields define how the app will be labeled on the device. The start_url specifies the page that will open when the app is launched. The display property is often set to "standalone" to make the app open in a fullscreen-like mode without browser UI elements. The theme_color and background_color enhance the user interface experience by defining the color schemes of the title bar and the background screen. The icons array provides different image sizes so that devices can select the appropriate icon resolution for their screen.

Integration with HTML and Service Worker:

In the HTML file, the manifest is linked using the <link rel="manifest" href="manifest.json"> tag in the <head> section. The theme color is also defined using a

<meta name="theme-color"> tag to align the browser's UI with the app's branding. Alongside the manifest, a service worker script must be registered to provide offline support and caching. The service worker handles the installation of resources and intercepts network requests to serve cached content when offline.

Handling the Install Prompt:

Modern browsers emit a before install prompt event when the app meets the installation criteria. This event can be captured and used to display a custom install button. When the user clicks the install button, the app calls prompt() on the stored event, allowing the user to install the app. The response to the prompt is then handled to check whether the user accepted or dismissed the installation.

Code:https://github.com/Kingmaker-2/PWA-7.git

Output:
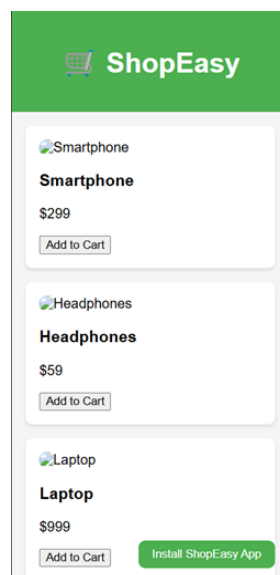
Install app Button
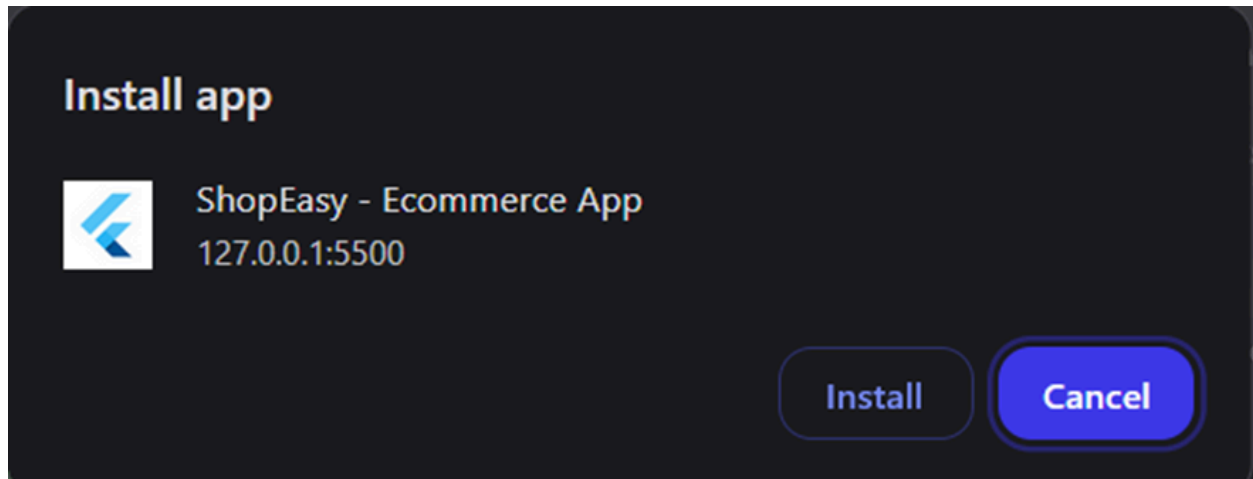
Chrome prompt to install app
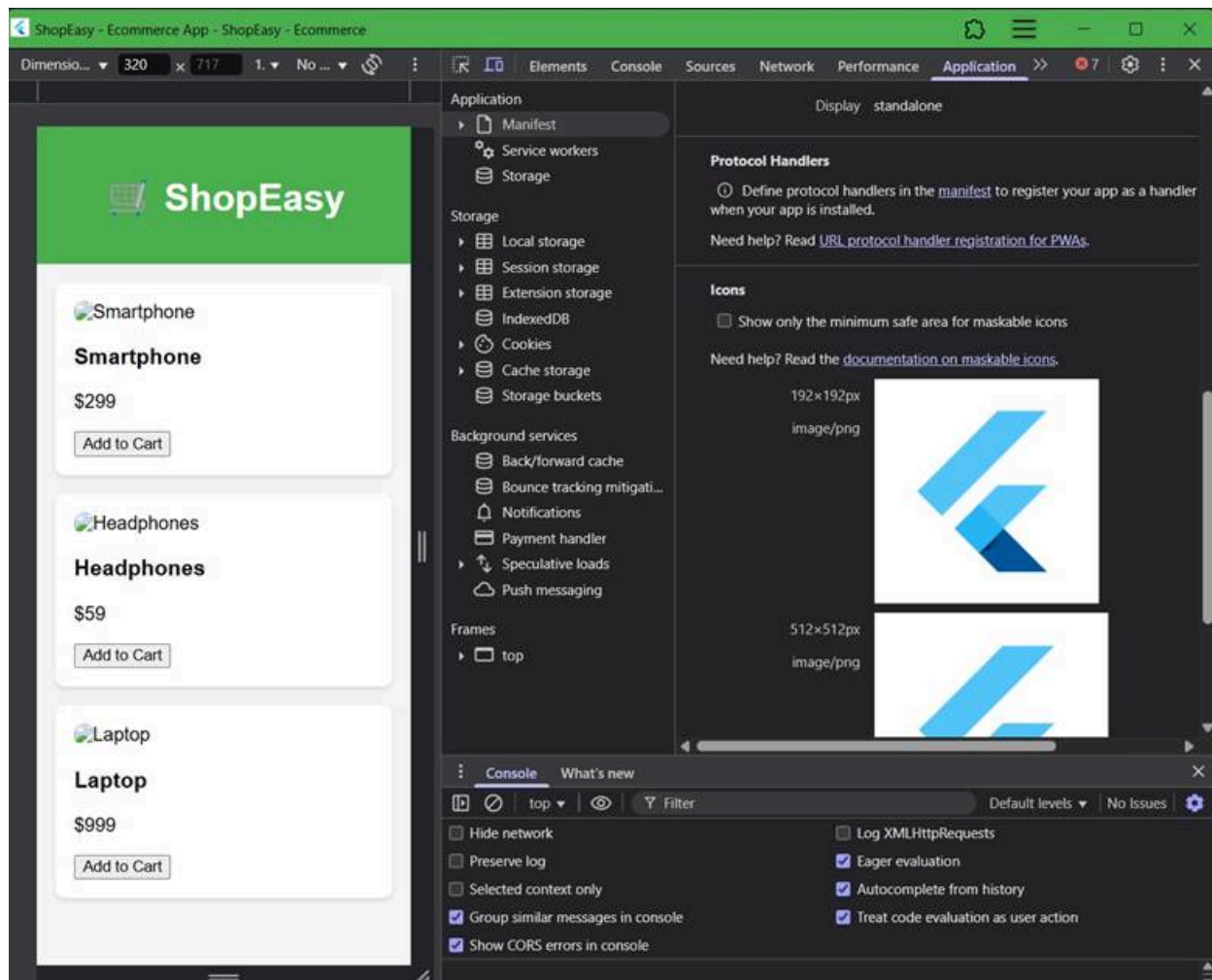
Installed app running

Conclusion: Implementing a manifest file and handling the install prompt are essential steps in transforming a traditional web application into a Progressive Web App. This setup not only enables the "Add to Homescreen" feature but also improves user engagement and accessibility by offering an app-like experience on any device. Through proper metadata configuration and service worker integration, web apps like the Ecommerce PWA "ShopEasy" can offer a seamless, installable experience to users.

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

Theory:

A Service Worker is a client-side script that runs in the background of a web browser, separate from the main browser thread, enabling features that do not require a web page or user interaction. It acts as a programmable network proxy, allowing developers to control how network requests from their web application are handled.

Service Workers are fundamental to Progressive Web Apps (PWAs), enabling key functionalities such as:

- Offline support by caching resources

- Background data synchronization

- Push notifications

They provide the ability to intercept and respond to network requests programmatically, which can improve application performance and user experience.

Service Worker Lifecycle:

The lifecycle of a Service Worker is distinct and different from typical JavaScript running in web pages. It involves several well-defined states and events:

a)   Registration: Before a Service Worker can operate, it must be registered by the web page. Registration is the process where the browser is told about the Service Worker script location, initiating its installation process.

b)   Installation:

- The install event is fired when the browser downloads the Service Worker script for the first time or when it detects a change in the script.

- During installation, the Service Worker typically caches static resources (HTML, CSS, JavaScript, images) to enable offline usage.

● The installation is considered successful only if the associated promises (such as caching resources) resolve successfully.

c)   Activation:

● After installation completes successfully, the Service Worker enters the activation phase.

● During activation, the Service Worker can clean up old caches from previous versions and take control of pages.

● The Service Worker only starts controlling pages after activation is complete.

d)   Idle/Waiting:

● If a previous version of the Service Worker is still controlling pages, the new Service Worker waits in the "waiting" phase until all pages controlled by the old version are closed.

● The new Service Worker activates only after the old one is no longer in use unless manually forced.

e)   Fetch:

● Once active, the Service Worker can intercept network requests made by the controlled pages.

● It can respond with cached resources or fetch updated resources from the network.

Detailed Explanation of Lifecycle Events:

● **Registration:**

Service Worker registration happens from within the main JavaScript thread of a web page, and typically looks like this:

if ('serviceWorker' in navigator) { navigator.serviceWorker.register('/sw.js')

.then(registration => {

console.log('Service Worker registered with scope:', registration.scope);

})

.catch(error => {

console.error('Service Worker registration failed:', error);

});

}

●   navigator.serviceWorker.register() tells the browser where the Service Worker file is located.

●   The registration returns a promise that resolves if registration succeeds or rejects on failure.

●   **Installation:**

Once registered, the browser attempts to install the Service Worker by firing the install event:

self.addEventListener('install', event => { event.waitUntil(

caches.open('static-cache-v1').then(cache => { return cache.addAll([

'/',

'/index.html', '/styles.css', '/script.js'

]);

})

);

});

●   The install event allows the Service Worker to prepare resources (usually by caching).

●   event.waitUntil() ensures that the Service Worker does not install until the caching completes successfully.

●   If the installation fails (e.g., cache promise rejects), the Service Worker will not install, and the browser may retry later.

●   **Activation:**

After installation, the Service Worker moves to the activation phase:

self.addEventListener('activate',   event   =>   {   event.waitUntil(caches.keys().then(cacheNames => {

return Promise.all(

cacheNames.filter(name => name !== 'static-cache-v1')

.map(name => caches.delete(name))

);

})

);

});

●   The activate event is an opportunity to remove outdated caches or perform other maintenance.

●   The Service Worker only starts controlling clients after this event completes.

●   Fetch Event (Interception of Network Requests)

After activation, the Service Worker can intercept network requests using the fetch event:

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

return response || fetch(event.request);

})

);

});

● The Service Worker first attempts to serve the requested resource from the cache.

● If not found, it fetches the resource from the network.

● This allows for offline capabilities and improved performance.

Code: https://github.com/Kingmaker-2/PWA-8.git

Output:
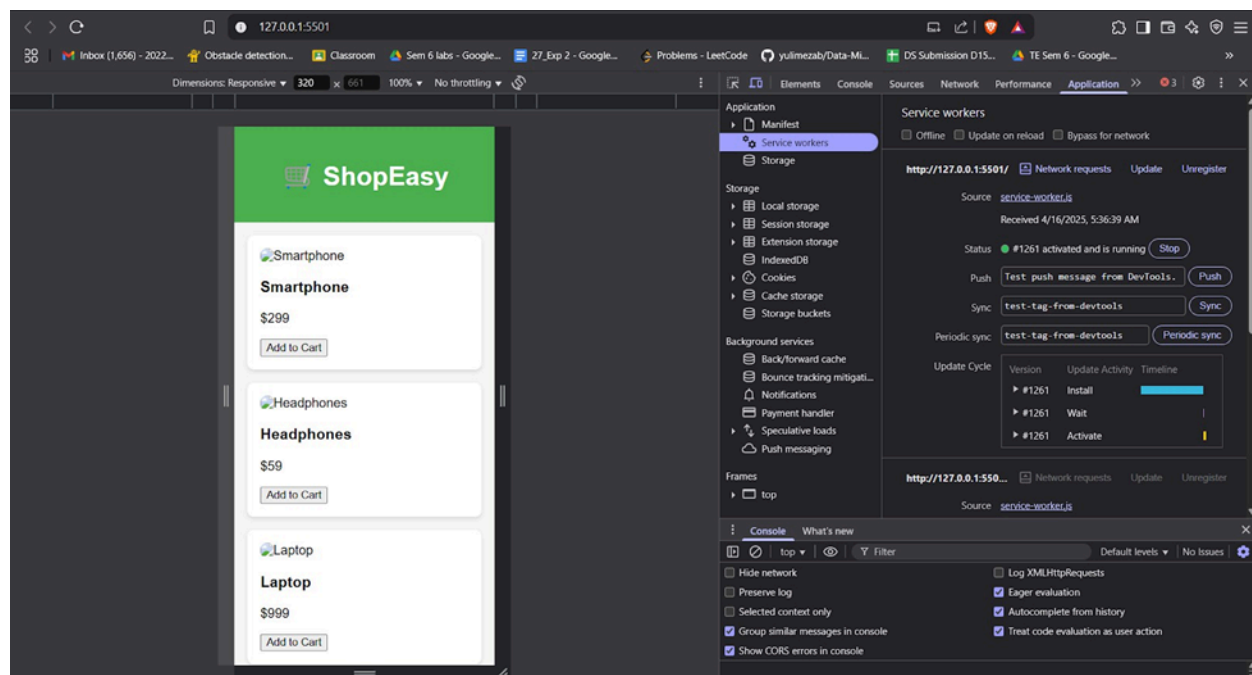
Developer tools showing the installation and working of Service Worker

Conclusion: Service Workers are a powerful feature of modern web browsers that enable developers to create reliable, fast, and engaging web applications by controlling network requests and managing resource caching. Understanding the Service Worker lifecycle—comprising registration, installation, activation, and fetch interception—is crucial for effective implementation. Through installation, essential resources can be cached to allow offline functionality, while activation ensures old caches are cleaned and the new Service Worker gains control. By intercepting

network requests, Service Workers provide significant performance improvements and resilience against unreliable network conditions. Mastery of Service Workers forms the foundation for building Progressive Web Apps that deliver a native-app-like user experience on the web.

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

Theory:

Service Workers are a foundational technology of Progressive Web Apps (PWAs), providing capabilities such as offline support, background synchronization, and push notifications. These capabilities are made possible through a set of built-in events that the Service Worker listens to and responds to. The most commonly used functional events include fetch, sync, and push. Each serves a unique purpose in enhancing web app reliability, responsiveness, and user engagement.

fetch Event:

Definition:

The fetch event is triggered every time a web page controlled by a Service Worker initiates a network request. This includes requests for HTML pages, CSS files, images, scripts, or API calls.

Why It's Important:

The fetch event allows developers to intercept and control outgoing network requests. This control is critical for enabling offline support, intelligent caching, and fallback mechanisms.

Use Cases:

- Serve resources from cache to improve load speed.

- Provide fallback pages when offline.

- Customize responses for different request types.

Sample Code:

self.addEventListener('fetch', function(event) { event.respondWith(caches.match(event.request).then(function(response) {

// If a cached version of the request exists, return it.

// Otherwise, fetch it from the network.

return response || fetch(event.request);

})

);

});

Explanation:

●   self.addEventListener('fetch', ...): Registers a listener for fetch events.

●   event.respondWith(): Tells the browser to use a custom response for this request.

●   caches.match(event.request): Searches the browser cache for a match to the current request.

●   fetch(event.request): Makes a network request if no cached version is found.

●   This approach is called the cache-first strategy, which prioritizes loading from cache and falls back to the network.

sync Event:

Definition:

The sync event is triggered when the browser regains network connectivity. It is used to execute deferred tasks (like sending data to the server) that couldn't complete due to being offline.

Why It's Important:

Many web apps require user actions (like form submissions) to be sent to the server. If the user is offline when this happens, the data can be temporarily stored and automatically synchronized once the device reconnects.

Use Cases:

- Retry failed form submissions or API calls.

- Sync data in the background without user interaction.

- Improve reliability of critical workflows.

Sample Code – Service Worker:

```
self.addEventListener('sync', function(event) { if (event.tag === 'sync-data') {
event.waitUntil(sendDataToServer());

}

});

function sendDataToServer() { return fetch('/submit-data', { method: 'POST',

body: JSON.stringify({ key: 'value' }), headers: {

'Content-Type': 'application/json'

}

});

}
```

Sample Code – Main JS Thread:

```
navigator.serviceWorker.ready.then(function(registration)          {          return
registration.sync.register('sync-data');

});
```

Explanation:

- The main script registers a sync event using registration.sync.register('sync-data').

● The Service Worker listens for the sync event with that tag.

● When triggered, the function sendDataToServer() is called to perform the actual data transfer.

● event.waitUntil() ensures the browser keeps the Service Worker alive until the task finishes.

● This ensures reliable data delivery, even if the user originally attempted it while offline.

push Event

Definition:

The push event is triggered when a server sends a push message to a client that has subscribed via the Push API. It enables the app to display notifications even when the web page is closed.

Why It's Important:

Push notifications allow web apps to maintain engagement with users by sending updates, alerts, or reminders — even when the app isn't currently open.

Use Cases:

● Send real-time updates or announcements.

● Notify users of background events (e.g., new messages).

● Deliver alerts while the app is closed or minimized.

Sample Code – Service Worker:

self.addEventListener('push', function(event) { const data = event.data ? event.data.json() : {};

const options = {

body: data.body || 'Default message body', icon: '/images/icon.png',

badge: '/images/badge.png'

};

event.waitUntil(

self.registration.showNotification(data.title || 'Default Title', options)

);

});

Explanation:

● event.data.json() parses the JSON payload received from the server.

● showNotification() uses the Notifications API to show the user a system-level notification.

● event.waitUntil() ensures the task completes before the Service Worker is terminated.

● This code ensures notifications can be received even if the app isn't currently open, which is a key feature of modern PWAs.

Code: https://github.com/Kingmaker-2/PWA-9.git

Output:
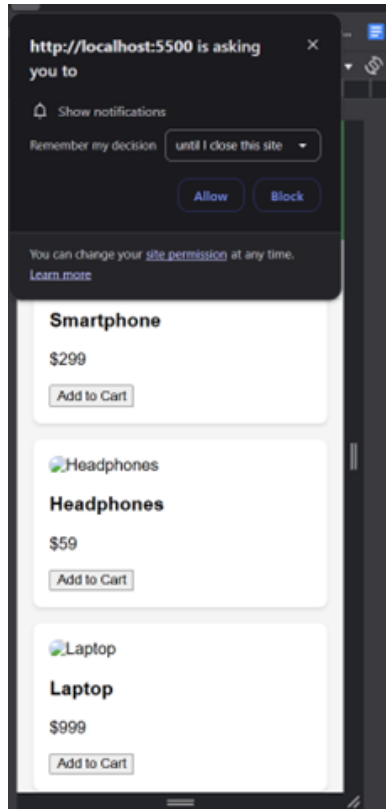
Request permission to send notifications
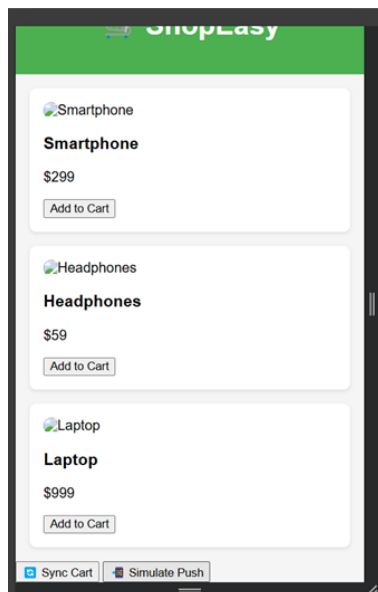
Push object received

Sync Notification

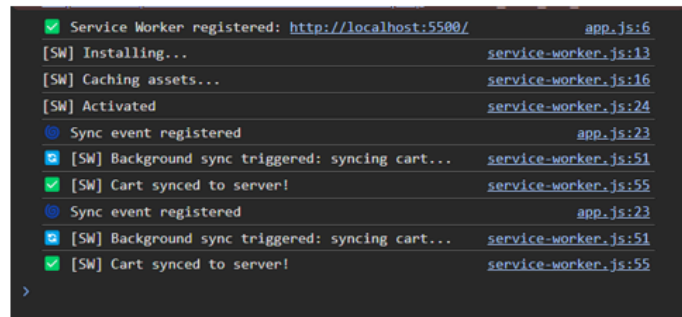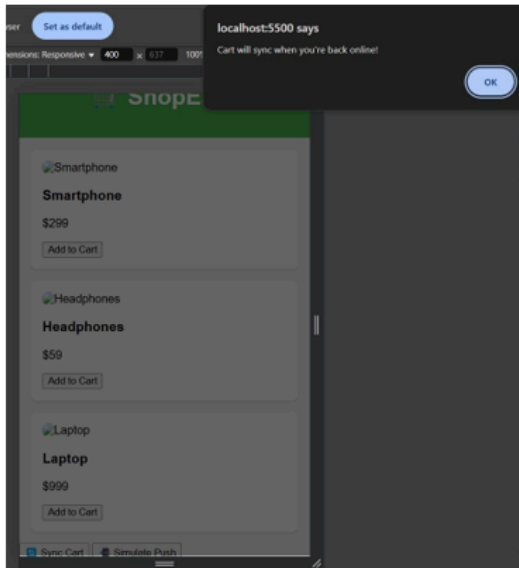**Conclusion:** Service Worker events such as fetch, sync, and push play a vital role in enhancing the functionality and reliability of modern web applications. The fetch event empowers developers to intercept and manage network requests, enabling offline capabilities and performance optimizations through caching strategies. The sync event ensures that important tasks, such as data submission or background synchronization,

are reliably completed even after temporary network interruptions. Meanwhile, the push event enables real-time communication by allowing servers to deliver updates and notifications to users, regardless of whether the application is open. Together, these events form the backbone of Progressive Web Apps (PWAs), providing seamless experiences across various network conditions and device states, and ultimately contributing to improved user

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

Theory:

Progressive Web Applications (PWAs) represent a modern approach to building web applications that combine the accessibility of websites with the rich user experience of native apps. Hosting these applications effectively is crucial to ensure performance, availability, and security. GitHub Pages offers a free and straightforward platform to host static web applications, including PWAs, making it a popular choice for developers worldwide.

What are GitHub Pages?

GitHub Pages is a static site hosting service integrated with GitHub repositories. It allows developers to publish web content directly from a repository without requiring any server-side infrastructure.

●Features of GitHub Pages:

○   Free and Public Hosting: GitHub Pages provides free hosting for open-source projects.

○   Automatic HTTPS: All sites hosted via GitHub Pages are served over HTTPS, ensuring security.

○   Branch or Folder Deployment: Content can be deployed from specific branches (like main or gh-pages) or designated folders (docs/).

○   Custom Domains: Supports connecting custom domain names.

○   Integrated with Git Version Control: Easy deployment and versioning through Git.

Understanding Progressive Web Applications (PWAs)

A Progressive Web Application (PWA) is a web app enhanced with modern web capabilities that enable it to provide a user experience similar to native applications.

●Characteristics of PWAs:

○   **Responsive:** Works well on any device and screen size.

○   **Offline Capabilities:** Uses service workers to cache resources and work without an internet connection.

○   **Installable:** Users can install the app to their home screen, bypassing app stores.

○   **Secure:** Served over HTTPS, which is mandatory for many PWA features.

○   **Discoverable:** Indexable by search engines.

○   **Re-engageable:** Supports push notifications.

●**Core Components:**

●   Manifest file (manifest.json): Describes the app's appearance, icons, and launch parameters.

●   Service Worker: Background script that manages caching and offline behavior.

●   Static assets: HTML, CSS, JavaScript files forming the app's UI and logic.

Why Host PWAs on GitHub Pages?

GitHub Pages offers a compelling hosting option for PWAs because:

●   Static Hosting Compatibility: PWAs are typically built into static files after a build process. GitHub Pages serves these static files efficiently.

●   HTTPS Support: Service workers require HTTPS for security reasons. GitHub Pages automatically provides HTTPS, enabling full PWA functionality.

●   Cost-Effective and Easy Deployment: GitHub Pages provides free hosting with minimal configuration, ideal for developers, learners, and small projects.

●   Seamless Integration with Git: Hosting from a GitHub repository enables easy version control and continuous deployment workflows.

General Steps to Deploy a PWA to GitHub Pages

Step 1: Prepare the Production Build

Use your framework's build tool to generate a production-ready version of the app. For

example:

npm run build

This generates a folder (build/, dist/, or equivalent) containing optimized static files.

Step 2: Configure Paths

Ensure paths in your PWA (such as those in manifest.json and asset references) are relative or correctly set according to the GitHub Pages URL, which usually includes your GitHub username and repository name.

Step 3: Setup GitHub Repository

Create or use an existing GitHub repository for your project.

Step 4: Choose Deployment Branch

GitHub Pages can deploy from the repository's main branch, a gh-pages branch, or a docs folder. For PWAs, it is common to deploy from the gh-pages branch.

Step 5: Deploy Static Files

There are multiple ways to deploy:

●   Manual deployment: Commit and push your build files to the deployment branch.

●   Automated deployment: Use tools like gh-pages npm package to deploy automatically:

npm install gh-pages --save-dev

Add deployment scripts in your package.json:

"scripts": {

"predeploy": "npm run build", "deploy": "gh-pages -d build"

}

Then run:

npm run deploy

Step 6: Access Your PWA

Visit your PWA at: https://<username>.github.io/<repository-name>/

Important Considerations for PWAs on GitHub Pages

● **Routing Issues:**

Since GitHub Pages is static, client-side routing (e.g., React Router) can cause 404 errors when users refresh or access deep-linked URLs. Solutions include:

● **Using hash-based routing.**

● **Adding a 404.html fallback redirect to the index page.**

● **Service Worker and Cache Management**

Careful configuration of service worker caching is important to ensure users get the latest updates without stale content.
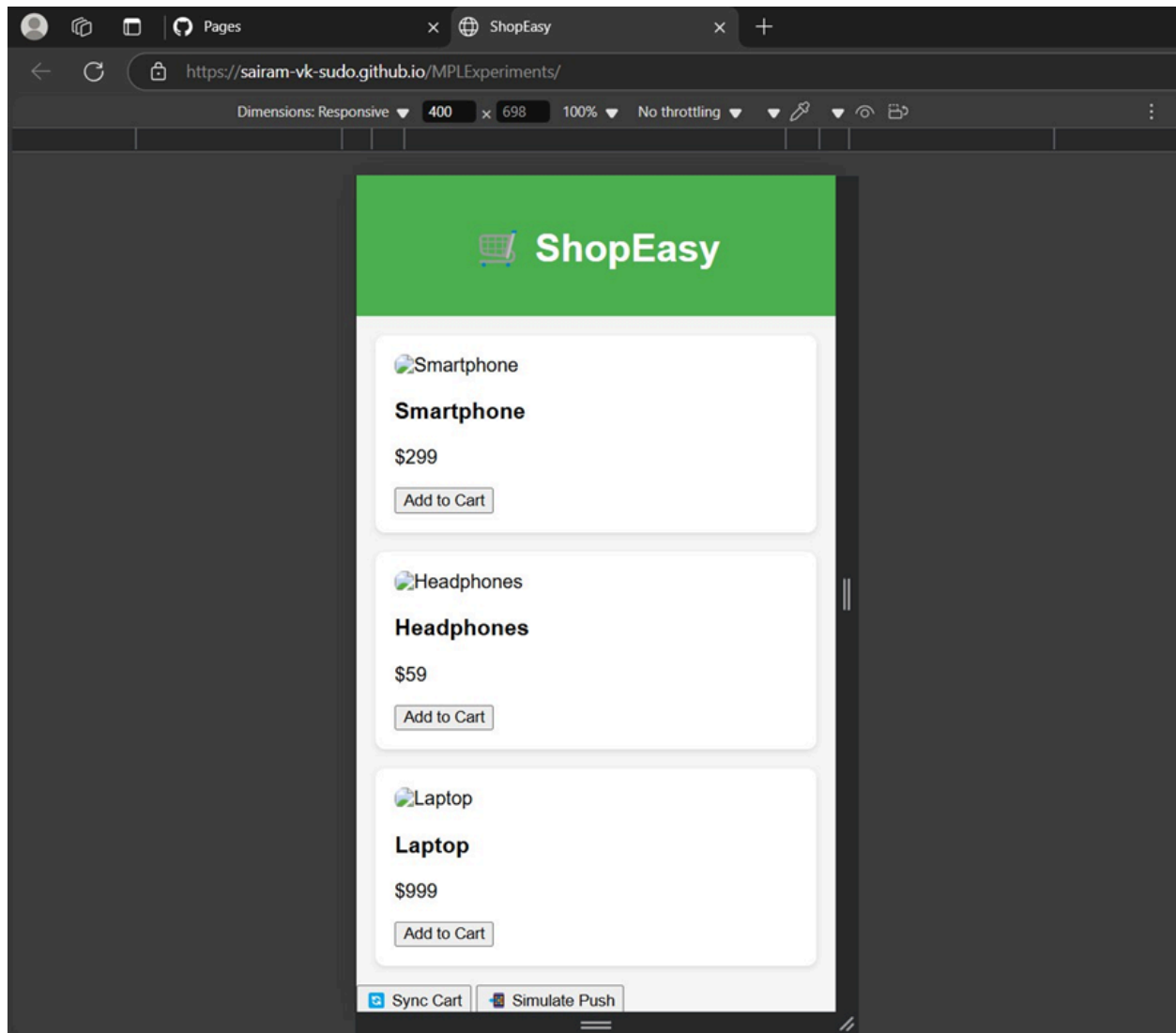
● **Asset Paths and Manifest Configuration**

Paths to assets like icons in manifest.json must reflect the GitHub Pages URL path structure.

● Limitations

○ No server-side code execution (e.g., no Node.js backend).

○ Not suitable for dynamic backend-powered apps unless combined with APIs.

Code:https://github.com/Kingmaker-2/PWA-10.git

Output:

Hosted App

Conclusion: Hosting Progressive Web Applications on GitHub Pages offers a practical, cost-effective, and secure solution for deploying modern web apps. GitHub Pages provides free static hosting with automatic HTTPS, which is essential for enabling core PWA features such as service workers and offline functionality. Its seamless integration with Git simplifies deployment and version control, making it accessible for developers

of all skill levels. While there are some limitations due to its static nature—such as the inability to run server-side code and potential routing challenges—these can often be managed with proper configuration. Overall, GitHub Pages is an excellent platform for delivering fast, reliable, and secure PWAs to a global audience with minimal setup and maintenance effort.

Aim: The aim of the experiment is to use Google Lighthouse PWA (Progressive Web App) Analysis Tool to test how well a Progressive Web App works. PWA is a type of web app that behaves like a native mobile app, with features like offline access, push notifications, and fast performance.

Theory:

Google Lighthouse is a free, automated tool developed by Google that helps you test the quality of web pages. It provides insights into performance, accessibility, SEO (Search Engine Optimization), and more.

When we use Lighthouse for PWA analysis, it helps us check:

1.   **Performance:** How Fast the PWA Loads and Responds

Performance measures how quickly your app loads and becomes interactive. A fast PWA ensures a better user experience, especially on mobile devices, reducing bounce rates and improving engagement.

2.   **Accessibility:** How Easy it is for Everyone to Use the PWA, Including People with Disabilities

Accessibility ensures that your PWA is usable by everyone, including those with disabilities. This includes features like screen reader compatibility, keyboard navigation, and readable text contrast.

3.   **Best Practices:** Whether the App Follows Recommended Web Development Guidelines

Best practices check if the app follows modern security and performance standards. It ensures the app is secure (using HTTPS), efficient, and uses up-to-date web technologies.

4.   **SEO:** How Well the PWA Can Be Found and Ranked by Search Engines

SEO ensures that your PWA is optimized for search engines. This includes proper meta tags, mobile-friendliness, and structured HTML, helping the app rank higher in search results.

5.   **Progressive Web App Features:** If it Behaves Like a PWA (Offline Access, etc.)

PWA features check if the app includes core characteristics like offline support via service workers, a web app manifest, and the ability to install the app on a device's home screen for a native-like experience.

Steps to perform the experiment:

1.Open the website you want to test: First, open the web app or website that you want to test in Google Chrome.

2.Open Developer Tools in Chrome:

○   Press Ctrl + Shift + I on your keyboard (Windows/Linux) or Cmd + Option

+ I (Mac).

○   Or right-click anywhere on the page and click Inspect.

3.Run Lighthouse Audit:

○   In the Developer Tools panel, click on the Lighthouse tab.

○   Under Lighthouse, you'll see options like Performance, Accessibility, Best Practices, SEO, and PWA.

○   Check the box for PWA to test if the website behaves like a Progressive Web App.

○   Now, click the Generate Report button.

4.Wait for the Report: Google Lighthouse will now analyze the website, and this might take a few moments.

5.Review the Report:

○   Once the audit is finished, Lighthouse will provide a detailed report.

○   The report will show scores for each category (e.g., PWA, performance, etc.) on a scale from 0 to 100.

○   For PWA, Lighthouse will specifically check features like:

■   Service Worker: Makes the app available offline.

■   Web App Manifest: Contains information like the app's name, icon, and theme color.

■   Add to Home Screen: If the user can install the app on their home screen like a native app.

6.Interpret the Results:

○   If your PWA score is high (above 90), it means the web app is well-optimized.

○   If the score is low, you can see specific suggestions for improvement, like adding a service worker or fixing the manifest file.

Code: https://github.com/Kingmaker-2/PWA-11.git
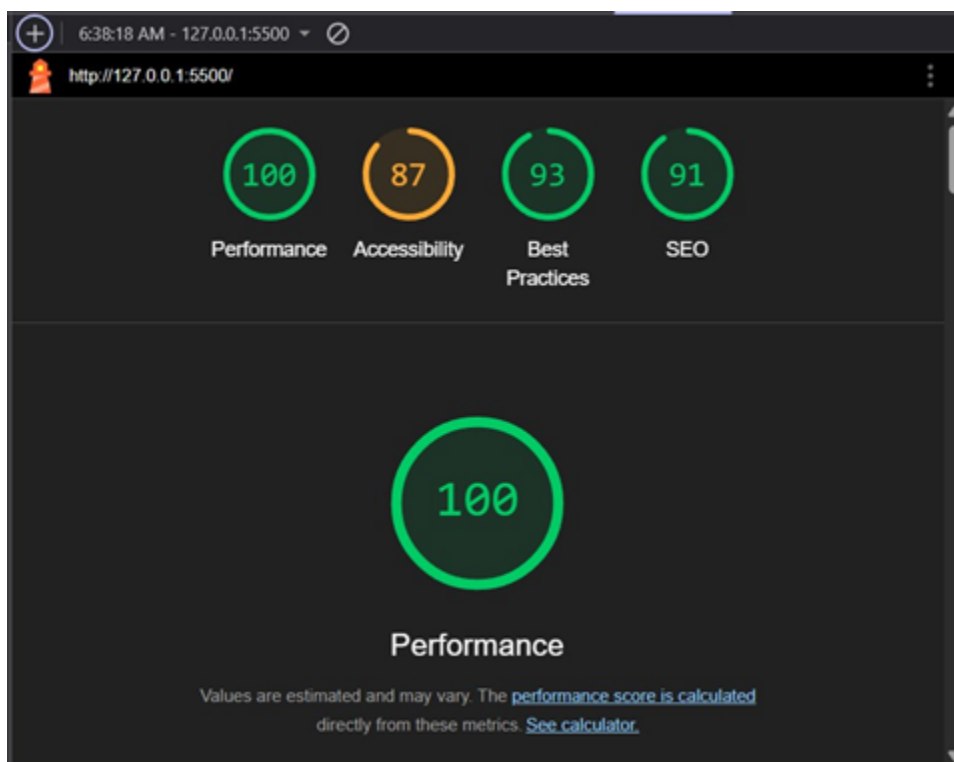
Output:

Google Lighthouse output

Conclusion: The Google Lighthouse PWA analysis tool provides a comprehensive evaluation of how well a web app performs as a Progressive Web App. By auditing key aspects such as performance, accessibility, best practices, SEO, and core PWA features, Lighthouse helps identify strengths and areas for improvement to deliver a seamless, native-app-like user experience. A high PWA score indicates that the app is optimized for offline use, fast loading, and easy installation, which enhances user engagement and accessibility across devices. Conversely, a low score highlights specific issues—such as missing service workers or manifest configurations—that developers can address to improve the app's reliability and usability. Overall, Lighthouse is an essential tool for developers aiming to build high-quality PWAs that meet modern web standards and prov