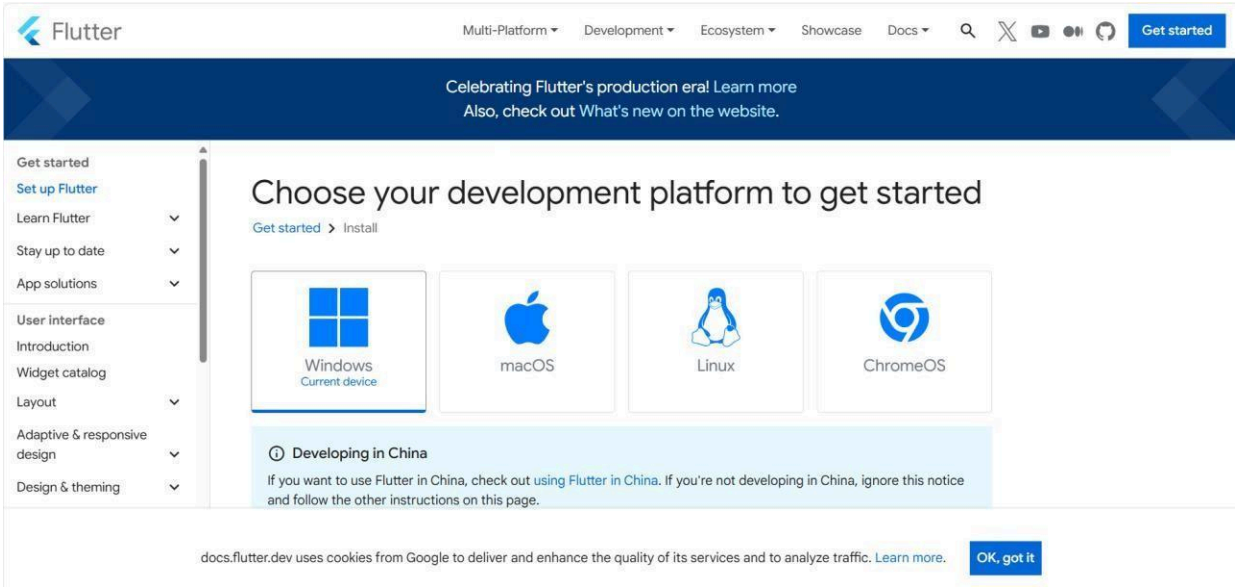
EXPERIMENT NO: - 01

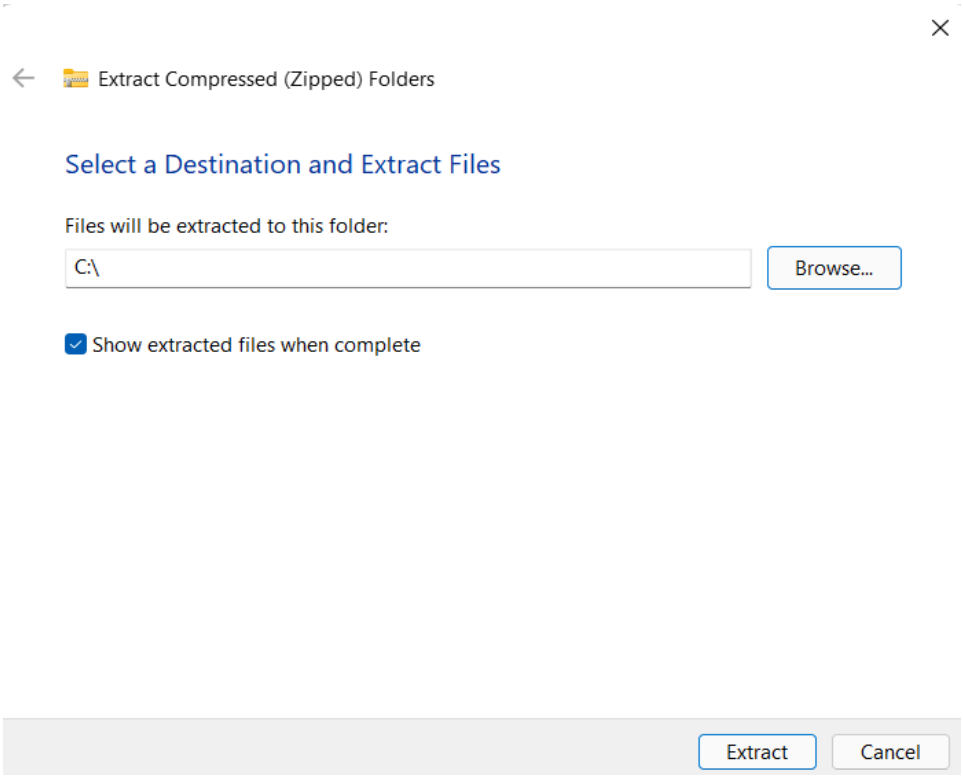
AIM: - Installation and Configuration of Flutter Environment.

Step 1: Go to the official Flutter website: https://docs.flutter.dev/get-started/install



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: Extract the zipped folder

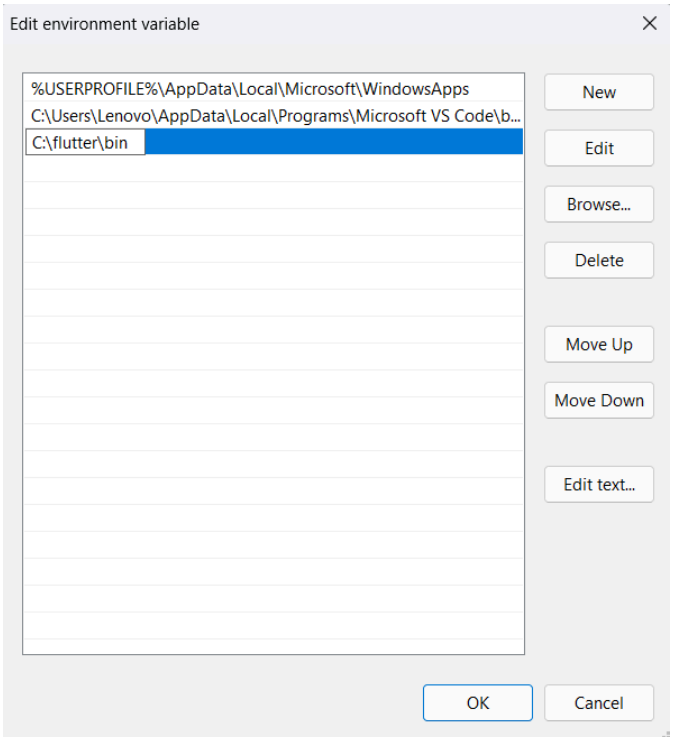

Extract Compressed (Zipped) Folders
Select a Destination and Extract Files
Files will be extracted to this folder:
C:\
Browse...
Show extracted files when complete
Extract
Cancel

Step 5 :- Add Flutter to System PATH

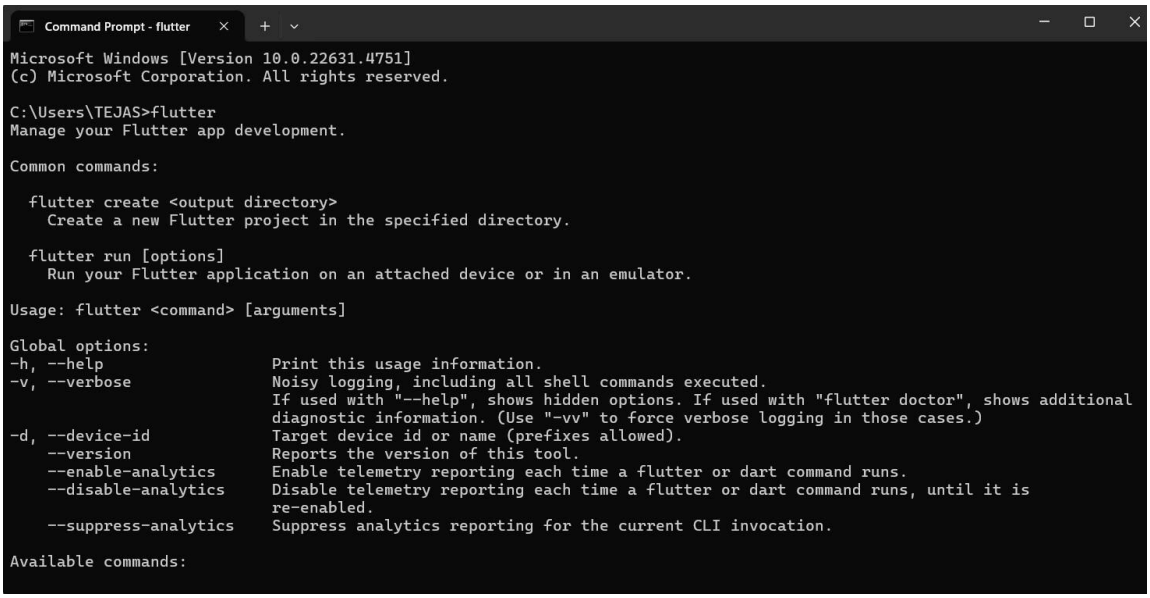
Right-click on the Start Menu > System > Advanced system settings > Environment Variables.

Under System Variables, find Path and click Edit.

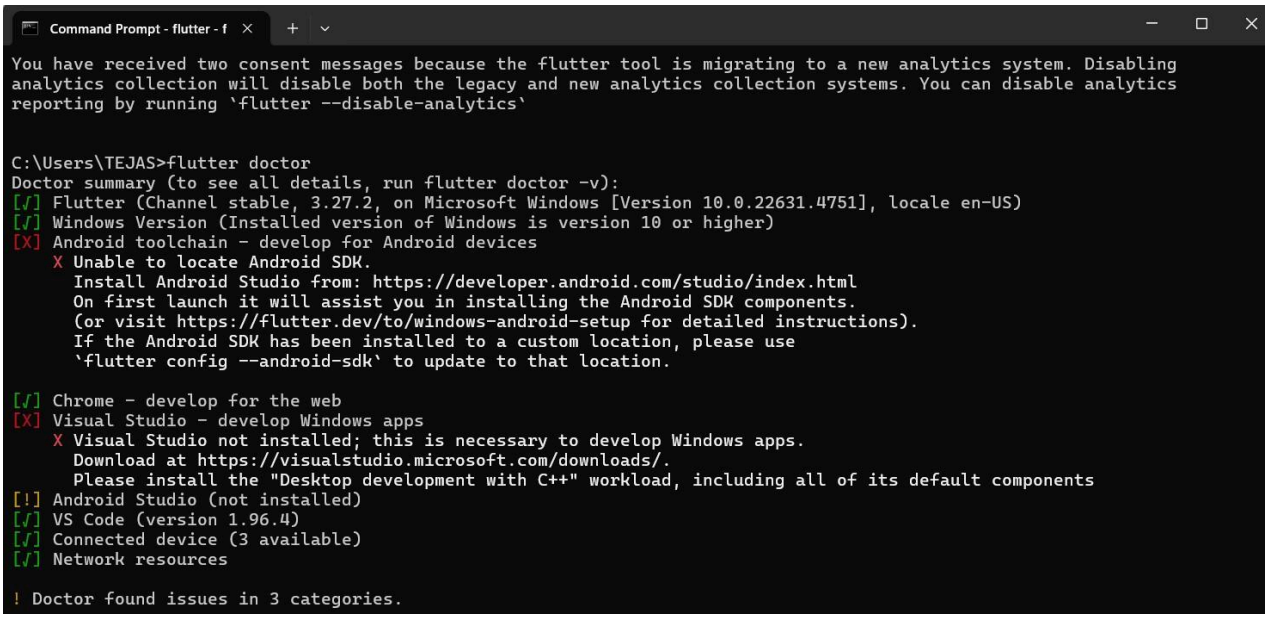
Add the full path to the flutter/bin directory (e.g., C:\flutter\bin).



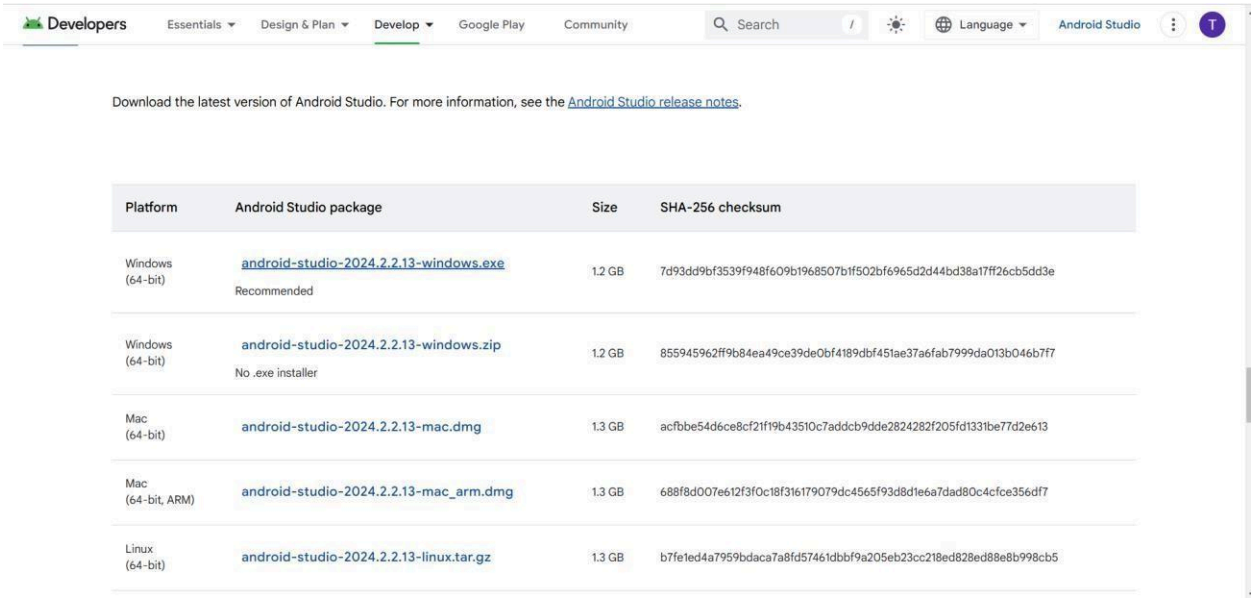
Step 6 : - Now, run the $ flutter command in command prompt.



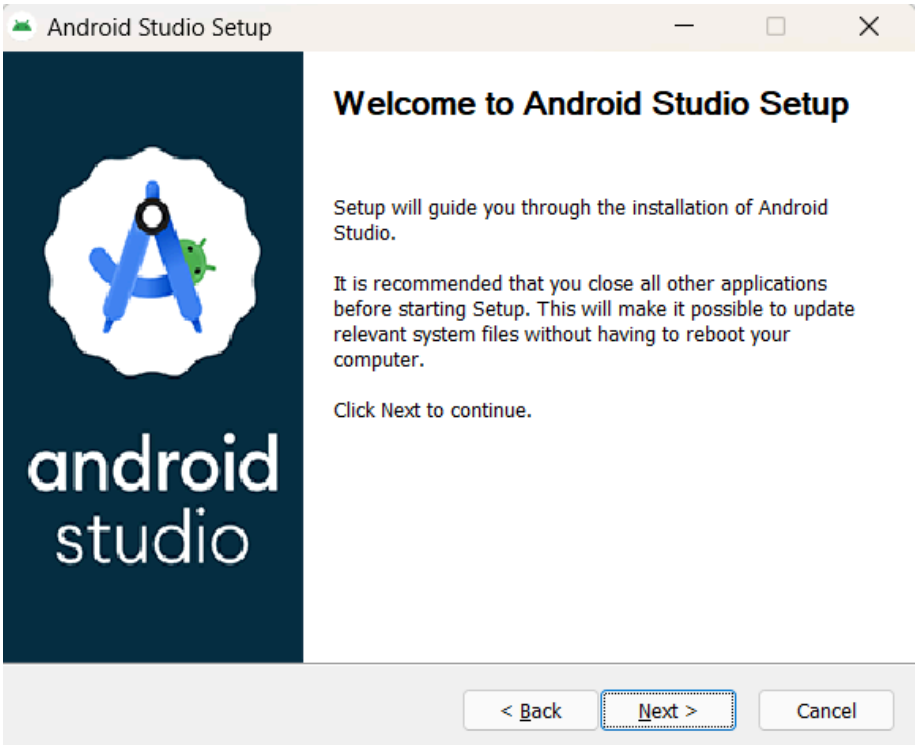
Step 7:- Run the $ flutter doctor command. This command checks for all the requirements of Flutter app development and displays a report of the status of your Flutter installation



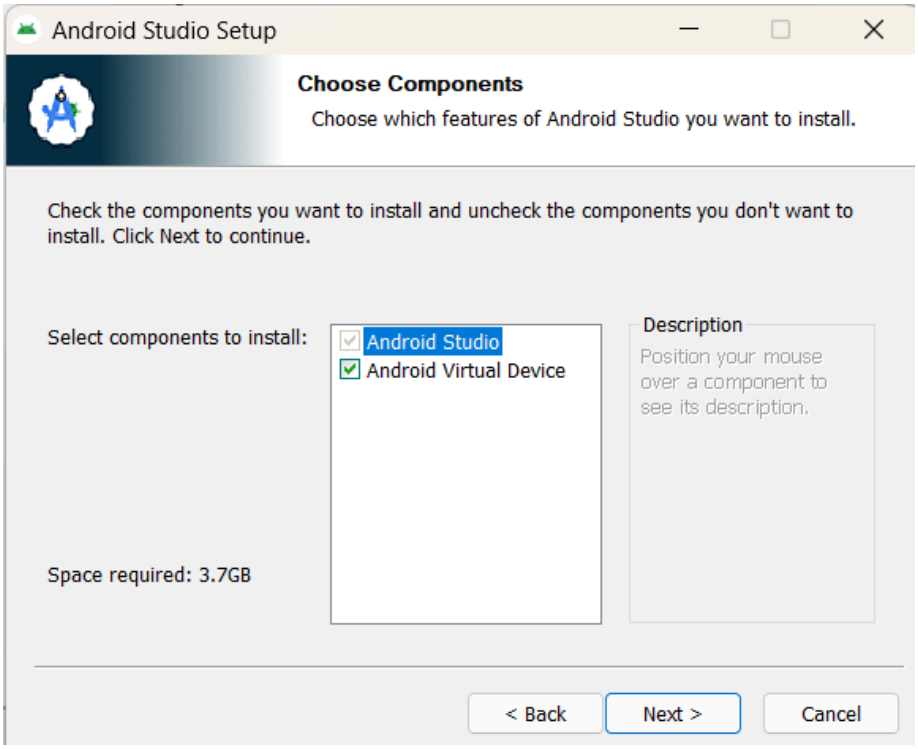
Step 8 : - Go to Android Studio and download the installer.



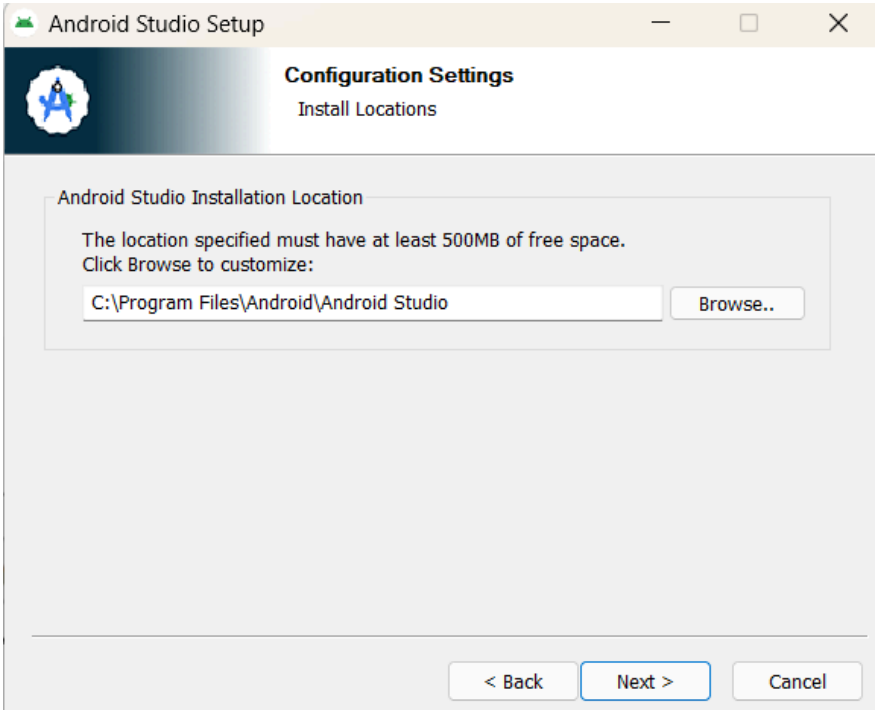
Step 8.1: - When the download is complete, open the .exe file and run it. You will get the following dialog box



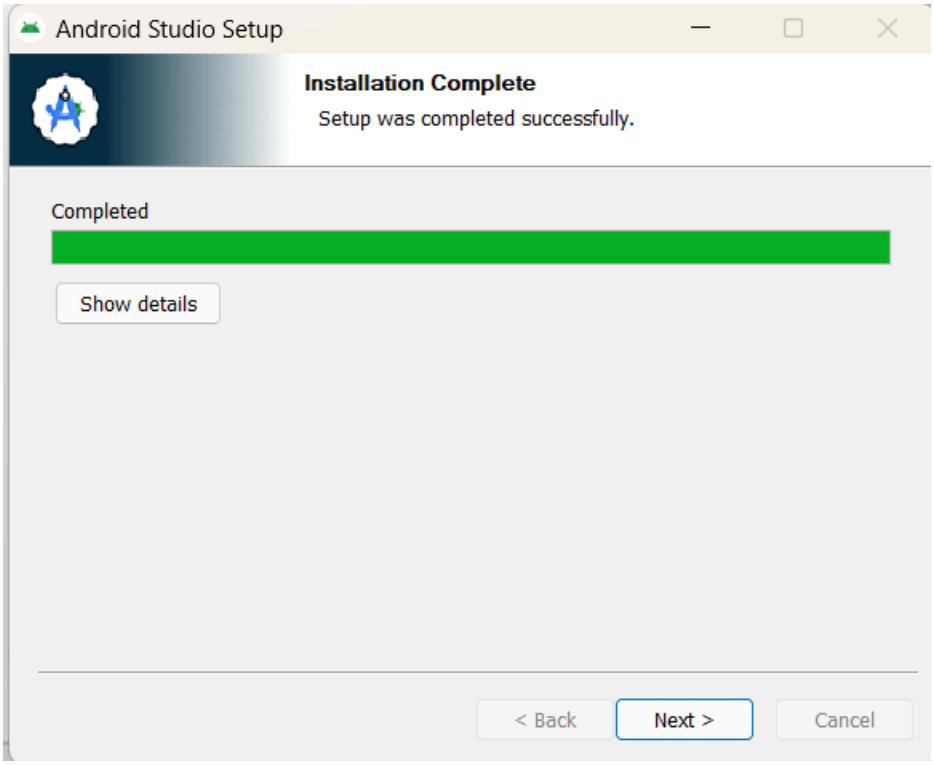
Step 8.2: - Select all the Checkboxes and Click on 'Next' Button.

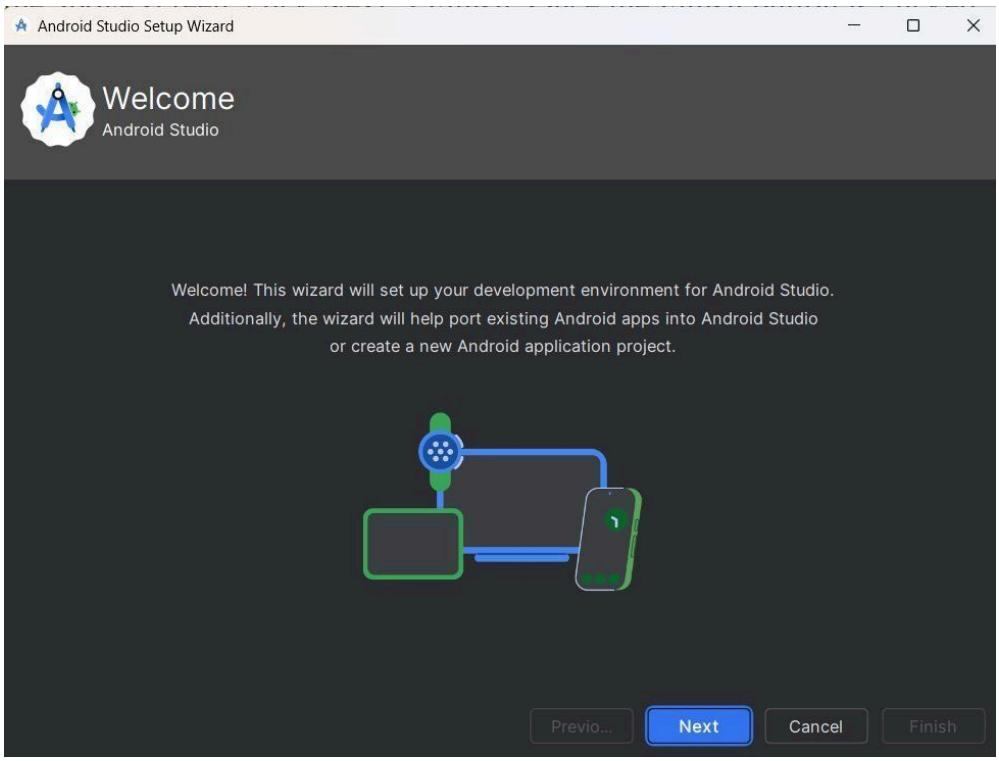


Step 8.3: - Change the destination as per your convenience and click on ‘Next’ Button.

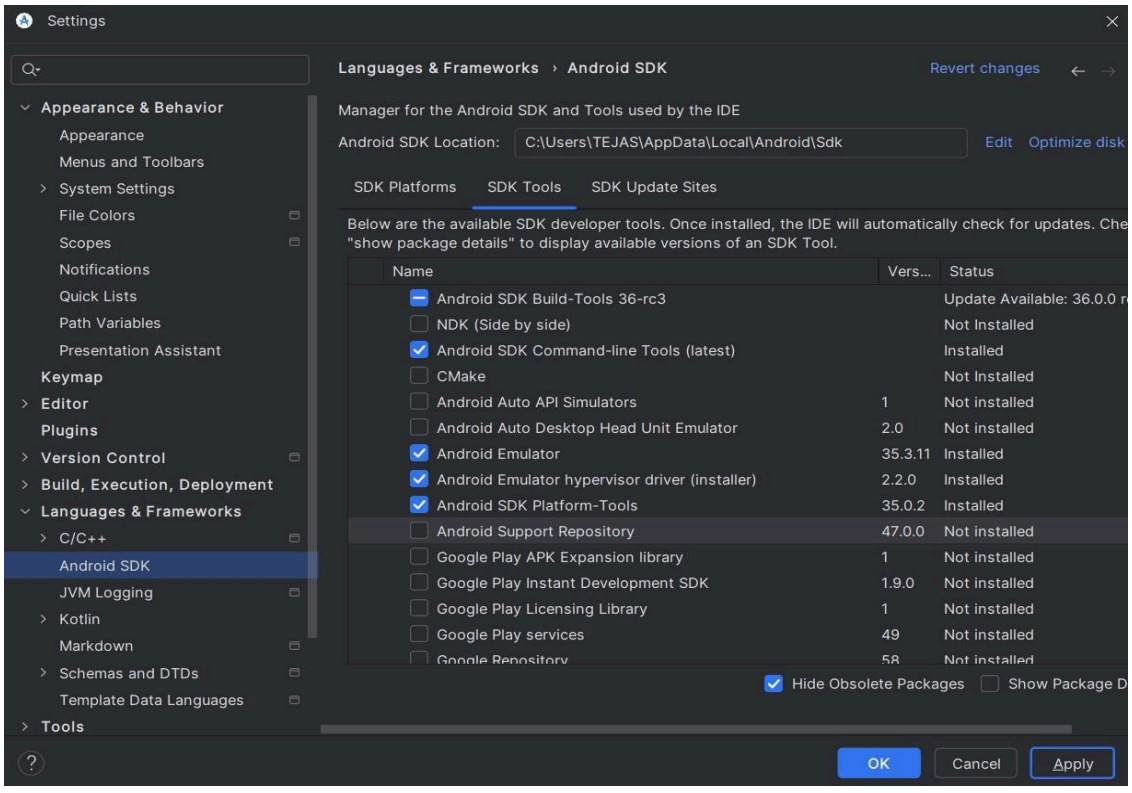


Step 8.4: - Follow the steps of the installation wizard. Once the installation wizard completes, you will get the following screen.

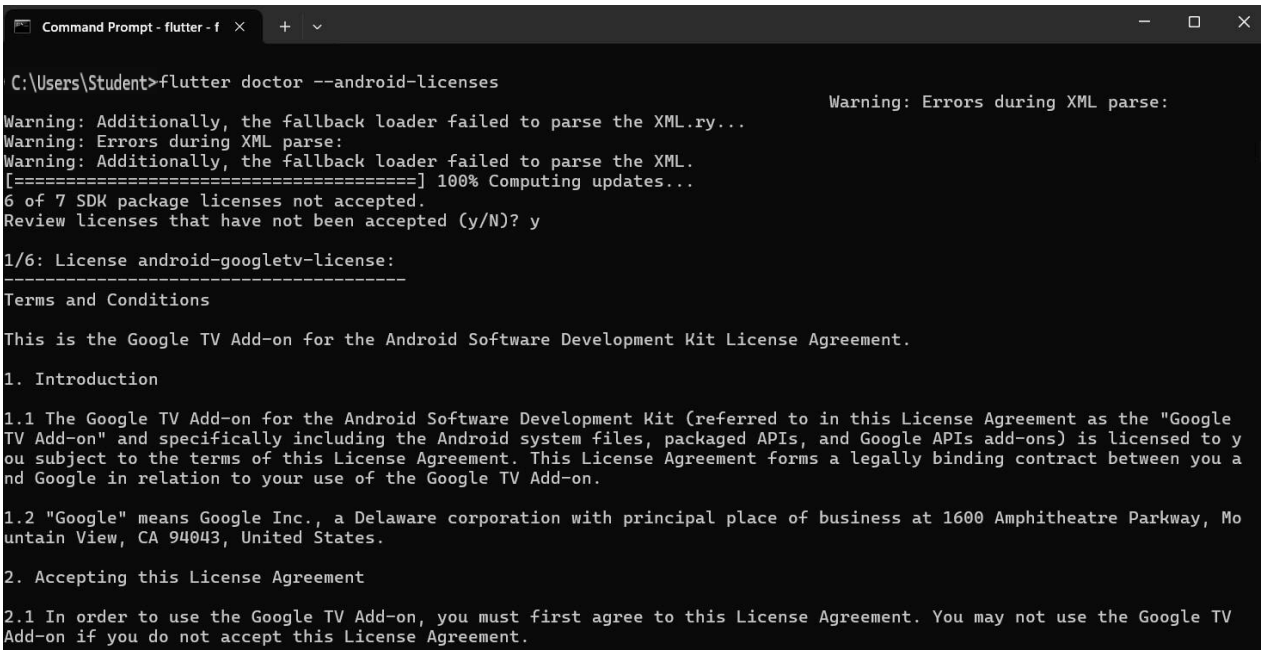


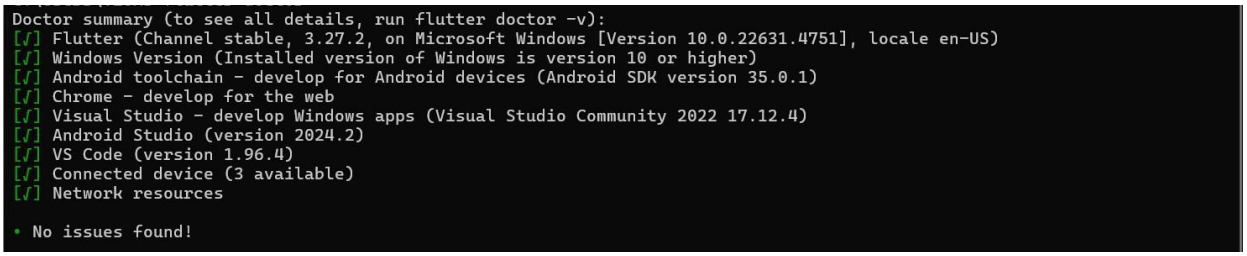


Step 8.5: - Go to Preferences > Appearance & Behavior > System Settings > Android SDK. Select the SDK Tools tab and check Android SDK Command-line Tools and Install it.

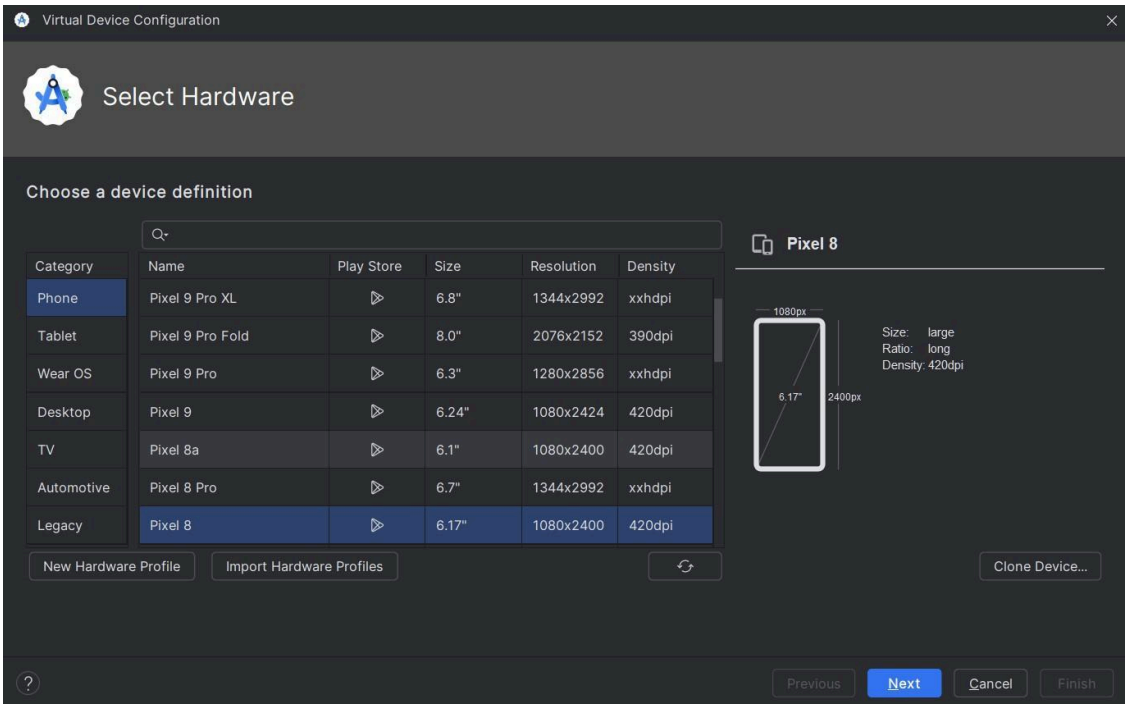


Step 9: - Open a terminal and run the following command

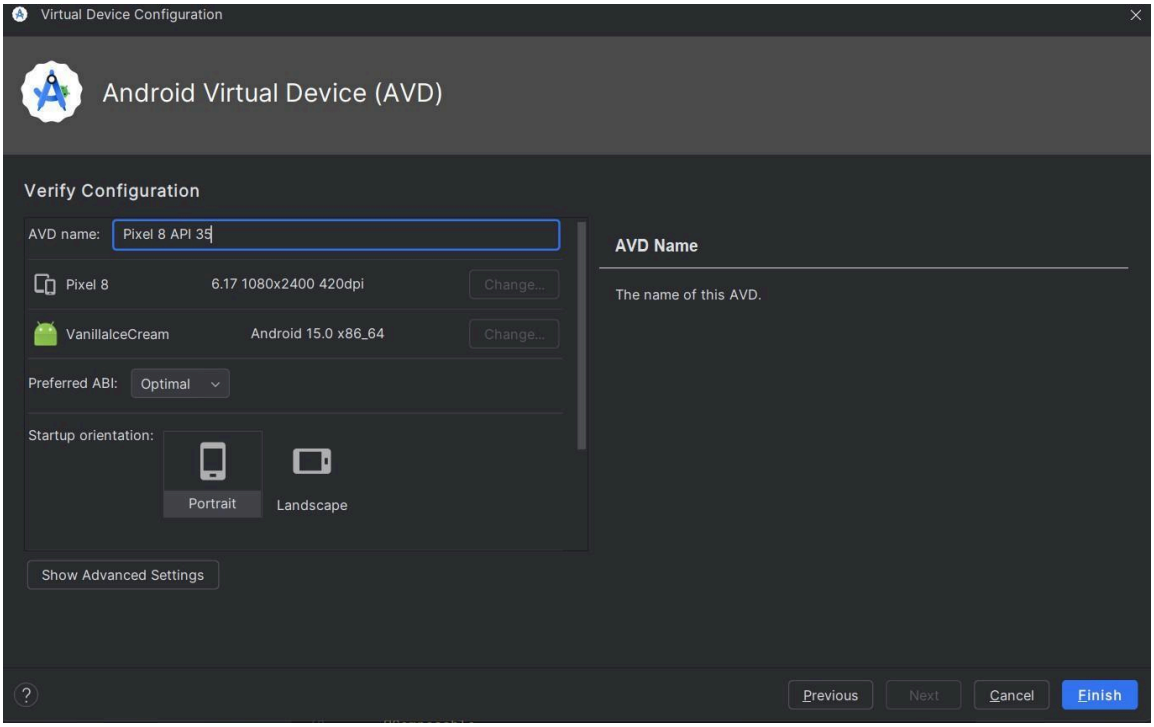




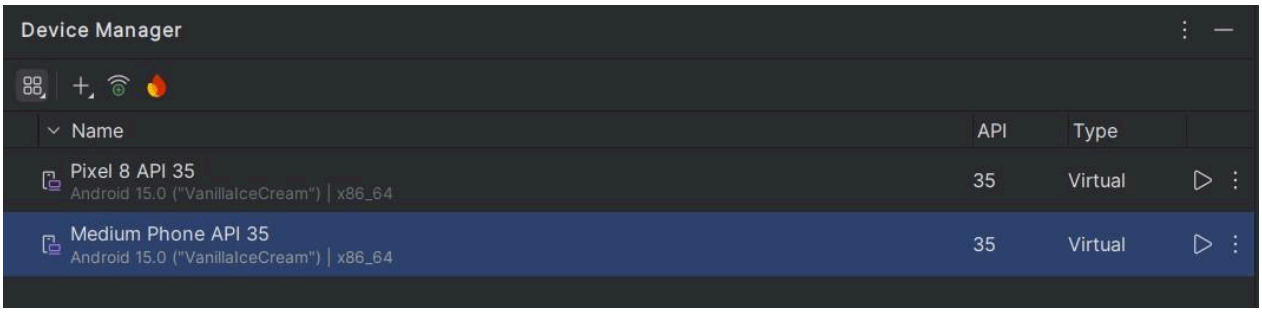
Step 10: - Next, you need to set up an Android emulator. It is responsible for running and testing the Flutter application

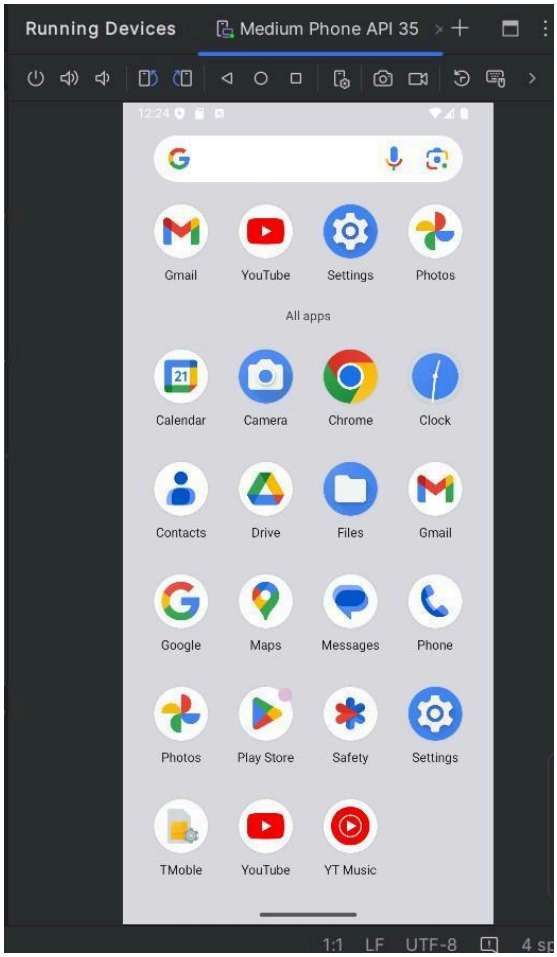


Step 10.1: - Open Android Studio and go to Tools > AVD Manager. Create a new virtual device.



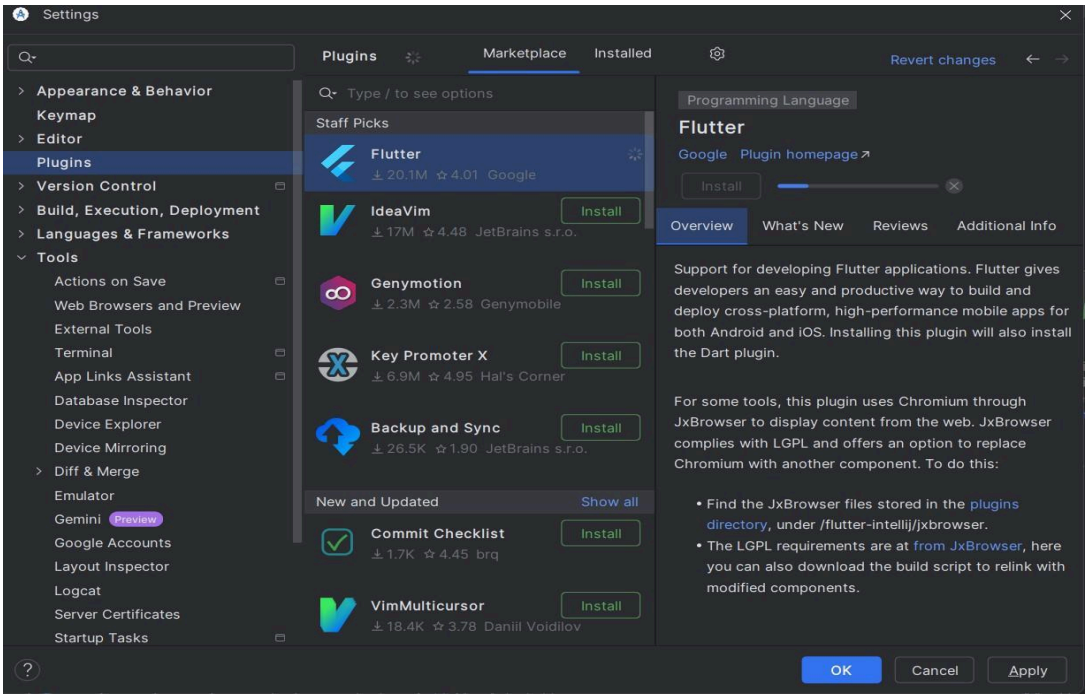
Step 10.2: - Click on the icon pointed into the red color rectangle. The Android emulator displayed as below screen

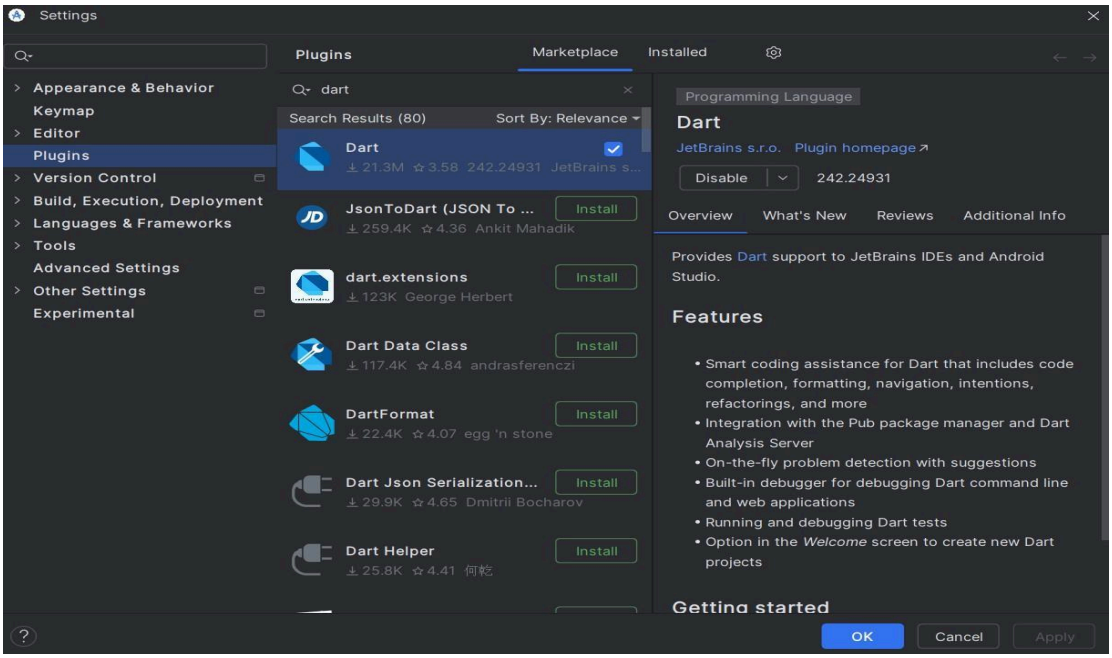




Step 11: - Now, install Flutter and Dart plugin for building Flutter application in Android Studio. These plugins provide a template to create a Flutter application, give an option to run and debug Flutter application in the Android Studio itself

Step 11.1: - Open the Android Studio and then go to File->Settings->Plugins. Now, search the Flutter plugin. If found, select Flutter plugin and click install

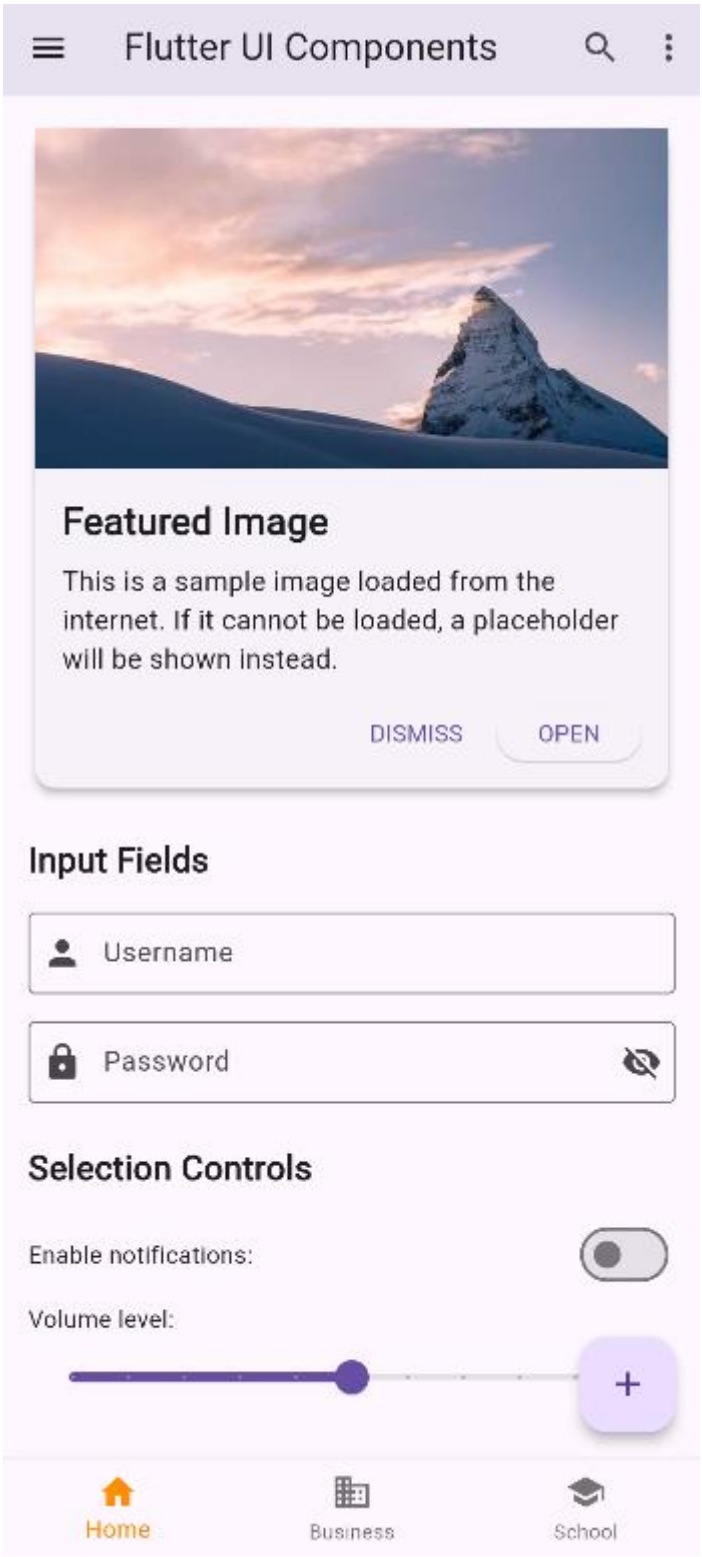




Jayan Karkera **D15B 28**

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

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



Usage of Widgets:

- Widgets like Row, Column, SizedBox, ElevatedButton, and Align are used to structure the UI.
- The SafeArea widget ensures that content is displayed within the safe area of the screen.
- The SingleChildScrollView widget allows scrolling when the content overflows the screen

List of Widgets

Flutter Scaffold
Flutter Container
Flutter Row & Column
Flutter Text
Flutter TextField
Flutter Buttons

Jayan Karkera MAD 3 D15B 28

- Including Icons in Flutter

Flutter provides built-in icons via the Icons class and allows the use of custom icons through the pubspec.yaml file.

1.1 Using Built-in Icons

Flutter provides an extensive set of material icons that can be used as follows:

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

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

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
return   MaterialApp(                home:
Scaffold(
        appBar: AppBar(title: Text("Flutter Icons")),
body: Center(           child: Icon(
Icons.favorite,             color: Colors.red,
size: 50.0,
          ),
        ),
      ),
    );
  } }
```

1.2 Using Custom Icons

To use custom icons, first, download or generate an icon font using FlutterIcon and add it to the pubspec.yaml file:

```
flutter:
fonts:     -
family:
CustomIcon
s       fonts:
        - asset: assets/fonts/custom_icons.ttf
```

Use it in your app:

Icon(IconData(0xe900, fontFamily: 'CustomIcons'))

- Including Images in Flutter

Flutter supports various ways to include images, such as from the assets folder, network URLs, and memory.

2.1 Adding Image Assets

1. Place images inside the assets/images/ directory.
2. Declare them in pubspec.yaml:

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

3. Use them in your app:

Image.asset('assets/images/sample.png', width: 200, height: 200)

2.2 Using Network Images

Load images directly from the internet:

Image.network('https://example.com/sample.jpg')

- Including Custom Fonts in Flutter

Custom fonts can enhance the UI by providing unique typography.

3.1 Adding Custom Fonts

1. Place font files in assets/fonts/.
2. Declare them in pubspec.yaml:

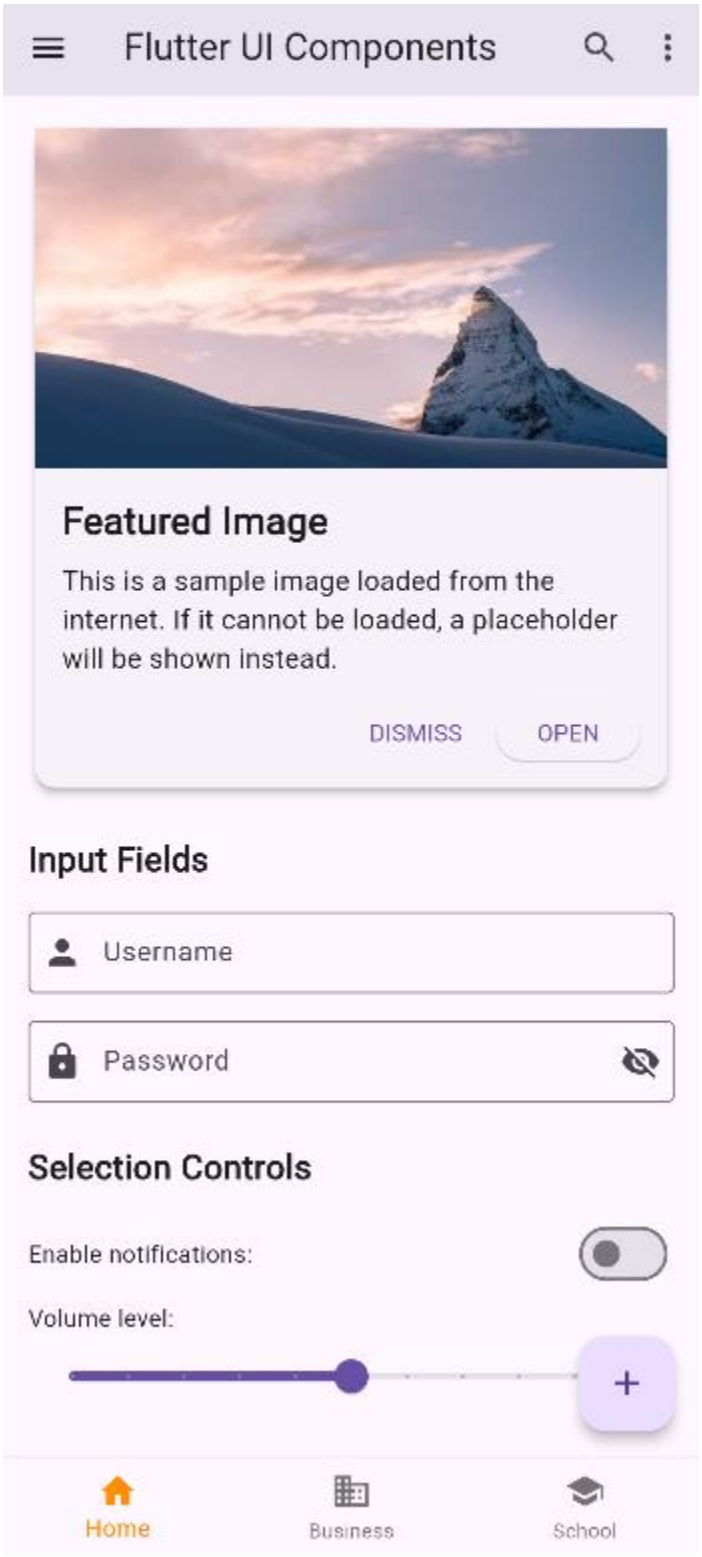
```
flutter:
fonts:
- family: CustomFont
    fonts:
- asset: assets/fonts/CustomFont-Regular.ttf
```

3.2 Using Custom Fonts in the App

Apply the font to text widgets:

```
Text(
  'Hello, Flutter!',
  style: TextStyle(fontFamily: 'CustomFont', fontSize: 24),
)
```

Output


Flutter UI Components
Featured Image
This is a sample image loaded from the internet. If it cannot be loaded, a placeholder will be shown instead.
DISMISS
OPEN
Input Fields
Username
Password
Selection Controls
Enable notifications:
Volume level:
Home
Business
School

Jayan Karkera
D15B 28

Lab 04

Aim: To create an interactive Form using form widget

Theory

Form validation is an essential feature in mobile applications to ensure the correctness of user inputs before processing them. In Flutter, `Form` and `TextFormField` widgets are used to create forms with built-in validation capabilities.

Key Concepts:

1. **GlobalKey:** Used to uniquely identify the form and manage its state.
2. **TextFormField:** A widget that allows users to enter input and supports validation.
3. **Validation Logic:** Defines conditions to verify the correctness of input.
4. **Submit Button:** Triggers form validation and processes the input if valid.

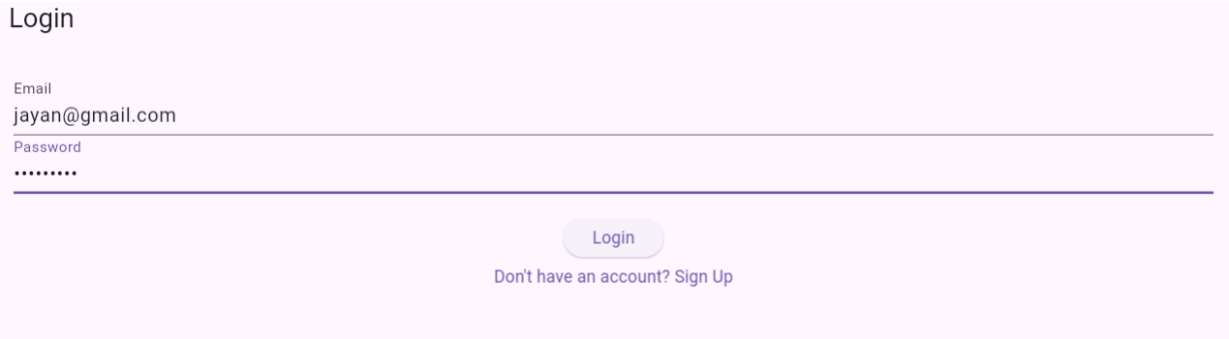
Code Implementation

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

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

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

```
class FormScreen extends StatefulWidget {
  @override
  _FormScreenState createState() => _FormScreenState();
}

class _FormScreenState extends State<FormScreen> {   final
_formKey = GlobalKey<FormState>();   final TextEditingController
_nameController = TextEditingController();

  void _submitForm() {
    if (_formKey.currentState!.validate()) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(content: Text('Form Submitted Successfully')),
      );
    }
  }

  @override
  Widget build(BuildContext context) {     return Scaffold(
appBar: AppBar(title: Text('Form Validation Demo')),       body:
Padding(          padding: const EdgeInsets.all(16.0),          child:
Form(           key: _formKey,           child: Column(             children: [
TextFormField(                 controller: _nameController,
decoration: InputDecoration(labelText: 'Enter your name'),
validator: (value) {                   if (value == null || value.isEmpty) {
return 'Name cannot be empty';
                }
return null;
},
              ),
              SizedBox(height: 20),
ElevatedButton(
onPressed: _submitForm,
child: Text('Submit'),
              ),
            ],
```

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

Output



Conclusion

Form validation in Flutter can be efficiently managed using the `GlobalKey`, `TextFormField`, and validation functions. This ensures user inputs are validated before processing, improving app reliability.

Jayan Karkera d15b 28

Aim: Navigation, Routing, and Gestures in Flutter

Introduction

Navigation, routing, and gestures are essential for building interactive Flutter applications. Navigation allows users to move between screens, routing manages structured transitions, and gestures detect user interactions like taps, swipes, and long presses.

Routing and Navigation in Flutter

In Flutter, routing and navigation are essential for managing screen transitions in an application. Flutter uses a stack-based navigation system, where screens (also called routes) are managed using a Navigator widget.

Types of Navigation in Flutter

1. Imperative Navigation (Navigator API)

- Uses Navigator.push() and Navigator.pop()
- Follows a stack-based approach (LIFO - Last In, First Out)
- Best suited for simple applications

2. Declarative Navigation (Go Router, Auto Route)

- Uses URL-based navigation
- Ideal for large applications with deep linking
- Navigator and Routes in Flutter

The Navigator manages the stack of screens in a Flutter app. Each screen is called a Route, and the Navigator widget helps in transitioning between routes.

1. Navigation in Flutter

Flutter's Navigator widget manages a stack of screens. Below is an example of basic navigation between two screens.

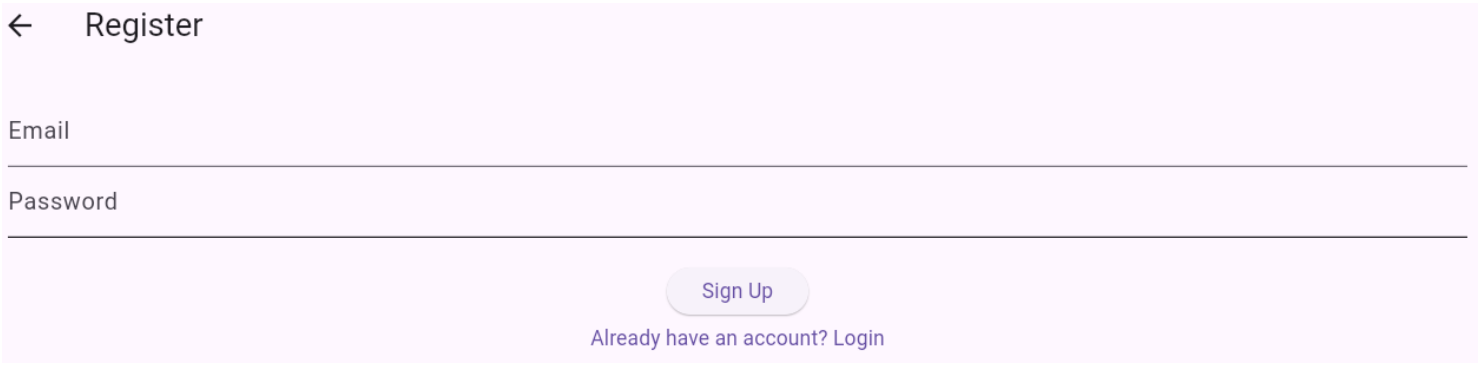
2. Gesture Detection in Flutter

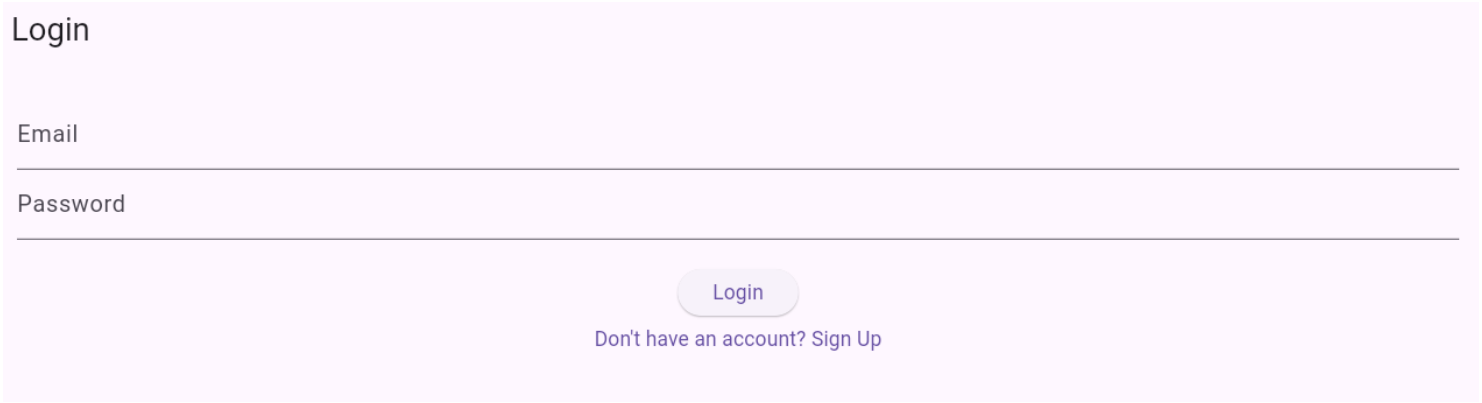
Flutter's GestureDetector widget captures various gestures like taps, double taps, long presses, and swipes.

OUTPUT

Conclusion

Navigation allows movement between screens using Navigator. Routing helps structure navigation better using named routes. Gesture detection enables interactivity with user input. These features make Flutter apps more user-friendly.





Expenses

massuri trip
Paid by: User1 - ₹15000
Split Type: percentage
jayan owes ₹7500.00
preksha owes ₹3000.00
jatin owes ₹4500.00

any trip
Paid by: User1 - ₹20000
Split Type: percentage
rush owes ₹2000.00
sid owes ₹1000.00
dhairyash owes ₹5000.00
jayan owes ₹6000.00
aryan owes ₹6000.00

eating chicken and all
Paid by: User1 - ₹15000
Split Type: equal
jatin owes ₹3750.00
preksha owes ₹3750.00
priyal owes ₹3750.00
jayan owes ₹3750.00

Add Expense

Amount

Description

Participants (comma-separated)

Equal Split

Add Expense

Rushabh Jain
D15B - 23

Setting Up Firebase with Flutter for iOS and Android Apps

This document provides a detailed step-by-step guide on how to integrate Firebase with a Flutter project for both iOS and Android platforms. Firebase offers a suite of tools for app development, including analytics, authentication, cloud storage, and more. By following this guide, you will be able to set up Firebase in your Flutter app and start using its features.

Prerequisites
Before starting, ensure you have the following:

Flutter SDK installed on your machine.

Android Studio or Xcode for Android and iOS development, respectively.

A Firebase account (create one at firebase.google.com).

A Flutter project created (flutter create project_name).

Step 1: Create a Firebase Project
Go to the Firebase Console.

Click Add Project.

Enter a project name and follow the prompts to create the project.

Once the project is created, you will be redirected to the Firebase project dashboard.

Step 2: Add Firebase to Your Flutter Project
For Android
In the Firebase Console, click the Android icon to add an Android app to your Firebase project.

Enter your app's details:

Android package name: Find this in your android/app/build.gradle file under applicationId.

App nickname (optional): Add a nickname for your app.

Debug signing certificate SHA-1 (optional): If you need Firebase Authentication or Dynamic Links, add your SHA-1 key.

Click Register App.

Download the google-services.json file and place it in the android/app directory of your Flutter project.

Add the following dependencies to your android/build.gradle file:

```
buildscript {
   dependencies {
      classpath 'com.google.gms:google-services:4.3.15' // Use the latest version
   }
}
```

Add the following to the bottom of your android/app/build.gradle file:

```
apply plugin: 'com.google.gms.google-services'
```

For iOS

In the Firebase Console, click the iOS icon to add an iOS app to your Firebase project.

Enter your app's details:

iOS bundle ID: Find this in your Xcode project under Bundle Identifier.

App nickname (optional): Add a nickname for your app.

Click Register App.

Download the GoogleService-Info.plist file.

Open your Flutter project in Xcode.

Drag and drop the GoogleService-Info.plist file into the Runner directory in Xcode.

Ensure the file is added to the Runner target.

Add the following to your ios/Podfile:

```
platform :ios, '11.0' # or higher
```

Run pod install in the ios directory to install Firebase dependencies.

Step 3: Add Firebase Dependencies to Flutter

Open your pubspec.yaml file in your Flutter project.

Add the following dependencies under dependencies:

```
dependencies:
  flutter:
    sdk: flutter
  firebase_core: latest_version # Required for Firebase integration
  firebase_analytics: latest_version # Optional: For analytics
  firebase_auth: latest_version # Optional: For authentication
  cloud_firestore: latest_version # Optional: For Firestore database
  firebase_storage: latest_version # Optional: For cloud storage
```

Run flutter pub get to install the dependencies.

Step 4: Initialize Firebase in Your Flutter App

Open your lib/main.dart file.

Import the Firebase Core package:

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

Initialize Firebase in the main function:

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

Step 5: Test Firebase Integration

Run your app on an Android or iOS emulator/device.

Check the Firebase Console to ensure your app is connected and sending data (e.g., analytics events).

Step 6: Use Firebase Services

Now that Firebase is set up, you can start using its services in your Flutter app. For example:

Firebase Authentication: Add user authentication using email/password, Google Sign-In, etc.

Firestore: Store and retrieve data from a NoSQL database.

Firebase Storage: Upload and download files.

Firebase Analytics: Track user behavior and app usage.

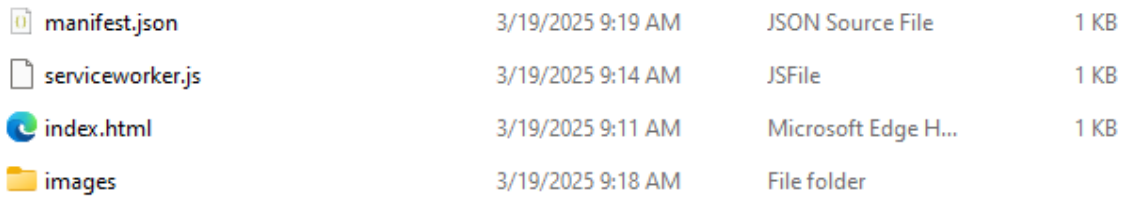
Troubleshooting
Android: If you encounter issues with the google-services.json file, ensure it is placed in the correct directory (android/app).

iOS: If the app crashes on launch, ensure the GoogleService-Info.plist file is added to the Xcode project and the Runner target.

Dependencies: Always use the latest versions of Firebase plugins and ensure there are no version conflicts.

Conclusion
You have successfully set up Firebase in your Flutter app for both iOS and Android platforms. You can now leverage Firebase's powerful features to enhance your app's functionality. For more details, refer to the official Firebase Flutter documentation.Replace latest_version with the actual version numbers of the Firebase plugins you are using. You can find the latest versions on pub.dev.

Jayan Karkera D15 B 28
MAD/PWA EXP 7



manifest.json	3/19/2025 9:19 AM	JSON Source File	1 KB
serviceworker.js	3/19/2025 9:14 AM	JSFile	1 KB
index.html	3/19/2025 9:11 AM	Microsoft Edge H...	1 KB
images	3/19/2025 9:18 AM	File folder	

//Index.html

```
<!DOCTYPE html>
<html lang="en">
<head>
  <meta charset="UTF-8" />
  <meta name="viewport" content="width=device-width, initial-scale=1.0" />
  <title>Expense Tracker</title>
  <link rel="manifest" href="manifest.json" />
  <link rel="stylesheet" href="styles.css" />
  <link rel="manifest" href="manifest.json" />
</head>
<body>
  <h1>Expense Tracker 💸</h1>
  <form id="expense-form">
    <input type="text" id="description" placeholder="Description" required />
    <input type="number" id="amount" placeholder="Amount" required />
    <button type="submit">Add Expense</button>
  </form>
  <ul id="expense-list"></ul>
  <h3>Total: ₹<span id="total">0</span></h3>

  <script src="app.js"></script>
</body>
</html>
```

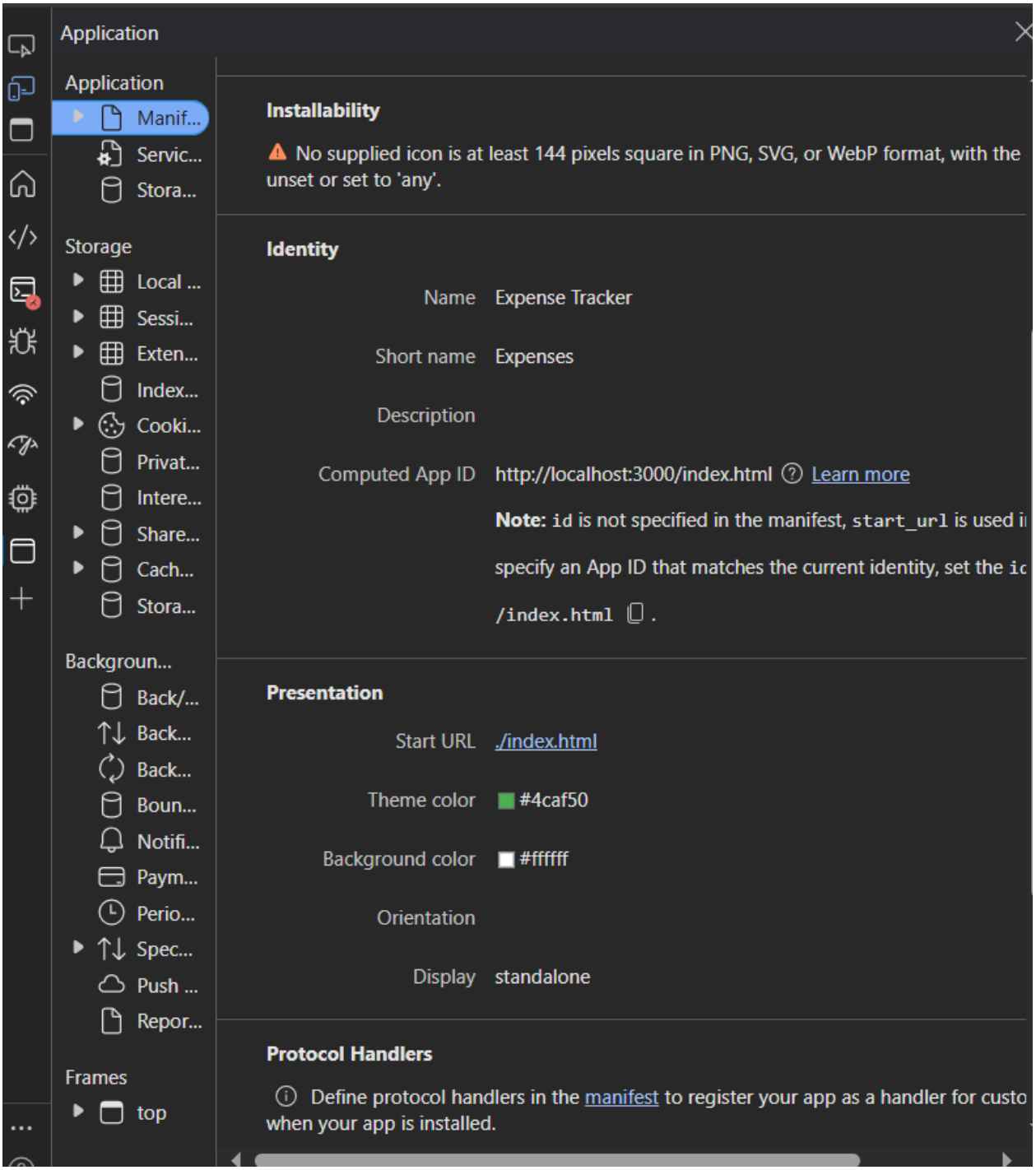
//manifest.json

```
{
    "name": "Expense Tracker",
    "short_name": "Expenses",
    "start_url": "./index.html",
    "display": "standalone",
    "background_color": "#ffffff",
    "theme_color": "#4caf50",
    "icons": [
      {
```

```
        "src": "icons/icon-192.png",
        "sizes": "192x192",
        "type": "image/png"
      },
      {
        "src": "icons/icon-512.png",
        "sizes": "512x512",
        "type": "image/png"
      }
    ]
  }
```

//serviceworker.js

```
const cacheName = 'expense-tracker-v1';
const filesToCache = [
  '/',
  '/index.html',
  '/styles.css',
  '/app.js',
  '/manifest.json',
  '/icons/icon-192.png',
  '/icons/icon-512.png'
];

// Install service worker
self.addEventListener('install', (e) => {
  e.waitUntil(
    caches.open(cacheName).then((cache) => {
      return cache.addAll(filesToCache);
    })
  );
});

// Activate service worker
self.addEventListener('activate', (e) => {
  console.log('Service Worker activated');
});

// Fetch data
self.addEventListener('fetch', (e) => {
  e.respondWith(
    caches.match(e.request).then((response) => {
      return response || fetch(e.request);
    })
  );
});
```


Application
Application
Manif...
Servic...
Stora...
Storage
Local ...
Sessi...
Exten...
Index...
Cooki...
Privat...
Intere...
Share...
Cach...
Stora...
Backgroun...
Back/...
Back...
Back...
Boun...
Notifi...
Paym...
Perio...
Spec...
Push ...
Repor...
Frames
top
Installability
No supplied icon is at least 144 pixels square in PNG, SVG, or WebP format, with the
unset or set to 'any'.
Identity
Name
Expense Tracker
Short name
Expenses
Description
Computed App ID
http://localhost:3000/index.html
Learn more
Note: id is not specified in the manifest, start_url is used i
specify an App ID that matches the current identity, set the ic
/index.html .
Presentation
Start URL
./index.html
Theme color
#4caf50
Background color
#ffffff
Orientation
Display
standalone
Protocol Handlers
Define protocol handlers in the manifest to register your app as a handler for custo
when your app is installed.

// SITE INSTALLED AS APP

Expense Tracker

Description

Amount

Add Expense

Total: ₹0

Service Worker Lifecycle

A service worker progresses through three key phases in its lifecycle:

1. Registration
2. Installation
3. Activation

Registration

Registration is the initial step where you inform the browser about your service worker. This process tells the browser where to find your service worker file and initiates its background installation. Here's an implementation example:

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

This code first checks if the browser supports service workers by testing for `navigator.serviceWorker`. It then registers the service worker using `navigator.serviceWorker.register()`, which returns a promise. On successful registration, it logs the scope, which defines which files the service worker can control. If registration fails, the error is logged.

By default, the service worker's scope is its location and all subdirectories. For example, if your service worker is in the root directory, it controls requests for all files on that domain.

You can also specify a custom scope:

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

Installation

When a service worker is registered, it triggers an installation event. You can listen for this event to perform tasks during installation, such as precaching resources for offline use:

```
// service-worker.js
self.addEventListener('install', function(event) {
  // Perform installation tasks
});
```

During installation, service workers often cache essential files to enable offline functionality and improve loading performance on subsequent visits.

Activation

After successful installation, the service worker enters the activation phase. If any pages are still controlled by a previous service worker, the new one enters a waiting state. It only activates when all pages using the old service worker are closed. This ensures only one service worker version runs at a time.

```
// service-worker.js
self.addEventListener('activate', function(event) {
  // Perform activation tasks like clearing old caches
});
```

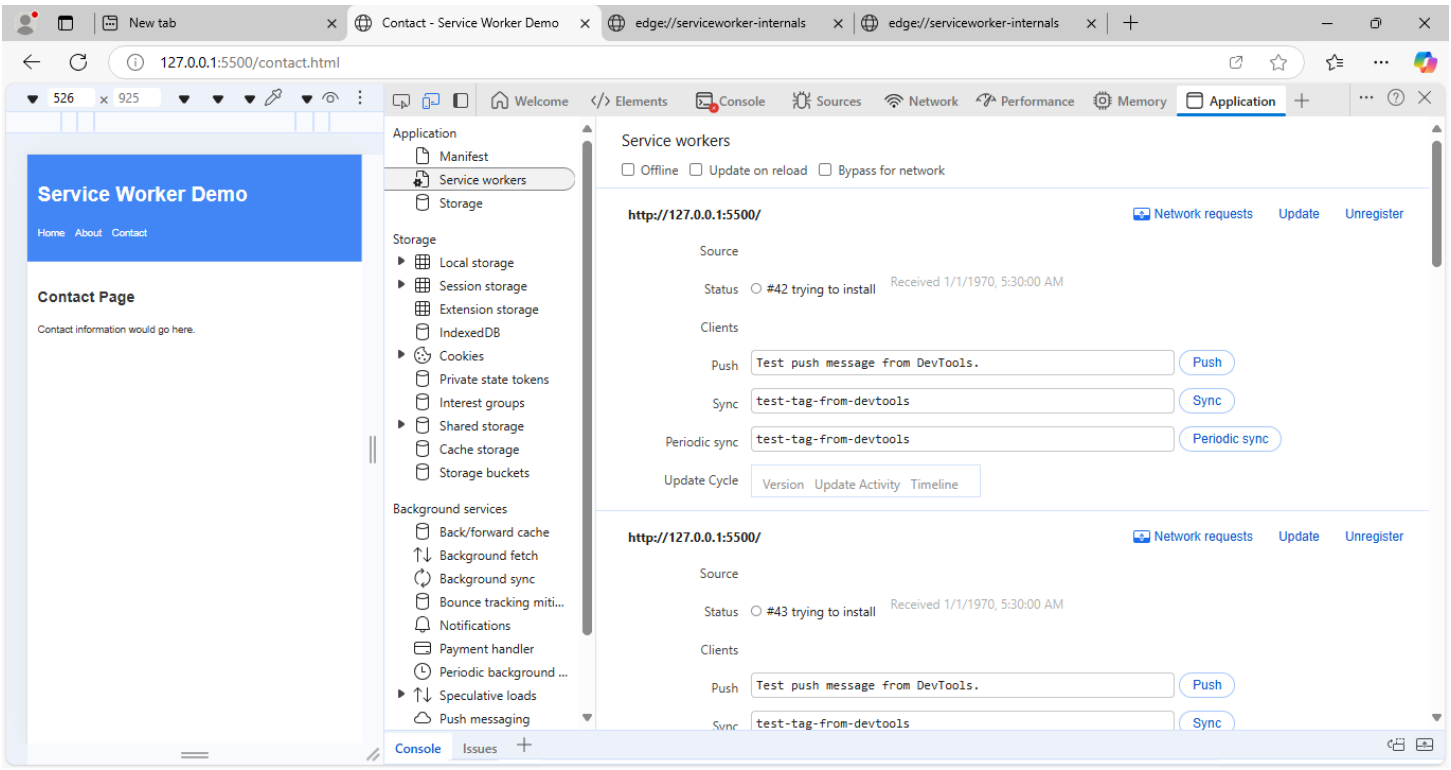
Code

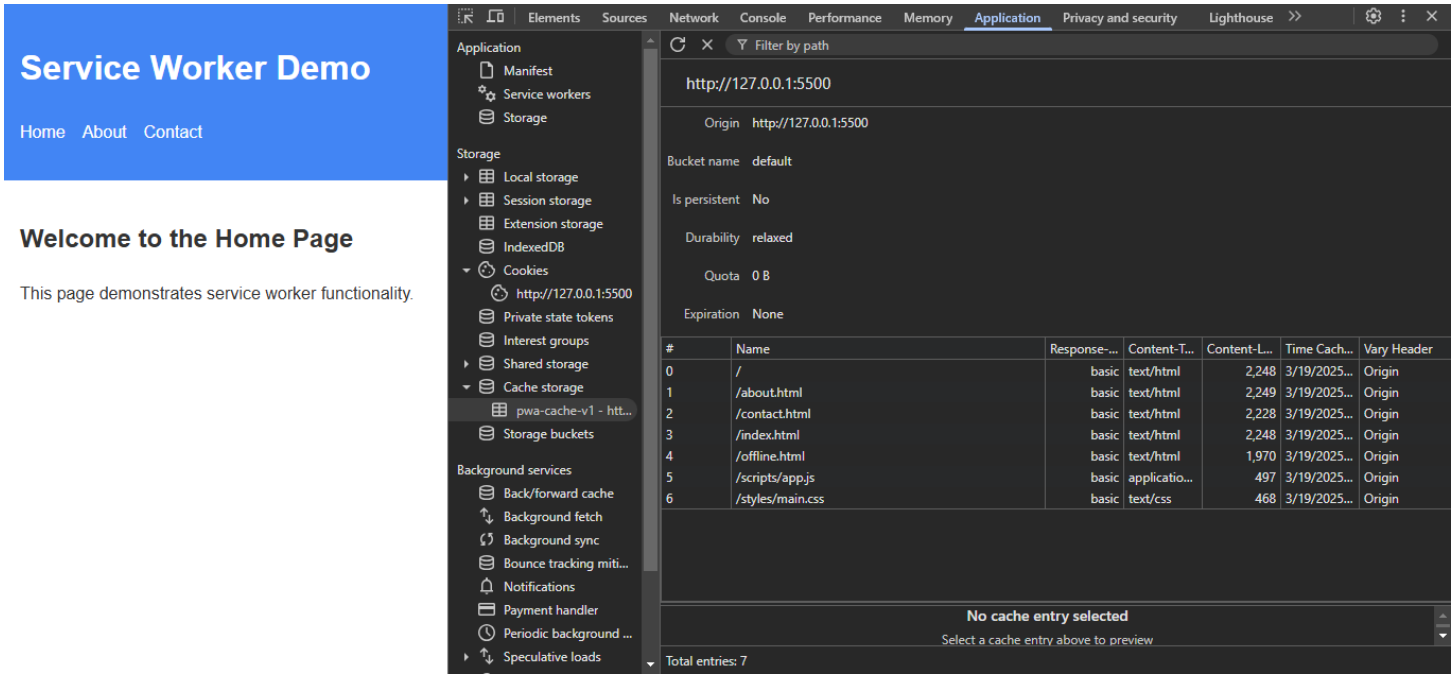
// service-worker.js

```
const cacheName = 'expense-tracker-v1';
const filesToCache = [
  '/',
  '/index.html',
  '/styles.css',
  '/app.js',
  '/manifest.json',
  '/icons/icon-192.png',
  '/icons/icon-512.png'
];

// Install service worker
self.addEventListener('install', (e) => {
  e.waitUntil(
    caches.open(cacheName).then((cache) => {
      return cache.addAll(filesToCache);
    })
  );
});

// Activate service worker
self.addEventListener('activate', (e) => {
  console.log('Service Worker activated');
});

// Fetch data
```

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

Output




Service Worker Demo
Home About Contact
Welcome to the Home Page
This page demonstrates service worker functionality.
Elements Sources Network Console Performance Memory Application Privacy and security Lighthouse
Application
Manifest
Service workers
Storage
Storage
Local storage
Session storage
Extension storage
IndexedDB
Cookies
http://127.0.0.1:5500
Private state tokens
Interest groups
Shared storage
Cache storage
pwa-cache-v1 - htt...
Storage buckets
Background services
Back/forward cache
Background fetch
Background sync
Bounce tracking miti...
Notifications
Payment handler
Periodic background ...
Speculative loads
Filter by path
http://127.0.0.1:5500
Origin http://127.0.0.1:5500
Bucket name default
Is persistent No
Durability relaxed
Quota 0 B
Expiration None
Name Response-... Content-T... Content-L... Time Cach... Vary Header
0 / basic text/html 2,248 3/19/2025... Origin
1 /about.html basic text/html 2,249 3/19/2025... Origin
2 /contact.html basic text/html 2,228 3/19/2025... Origin
3 /index.html basic text/html 2,248 3/19/2025... Origin
4 /offline.html basic text/html 1,970 3/19/2025... Origin
5 /scripts/app.js basic applicatio... 497 3/19/2025... Origin
6 /styles/main.css basic text/css 468 3/19/2025... Origin
No cache entry selected
Select a cache entry above to preview
Total entries: 7

Implementing Service Worker Events (Fetch, Sync, Push) for E-Commerce PWA

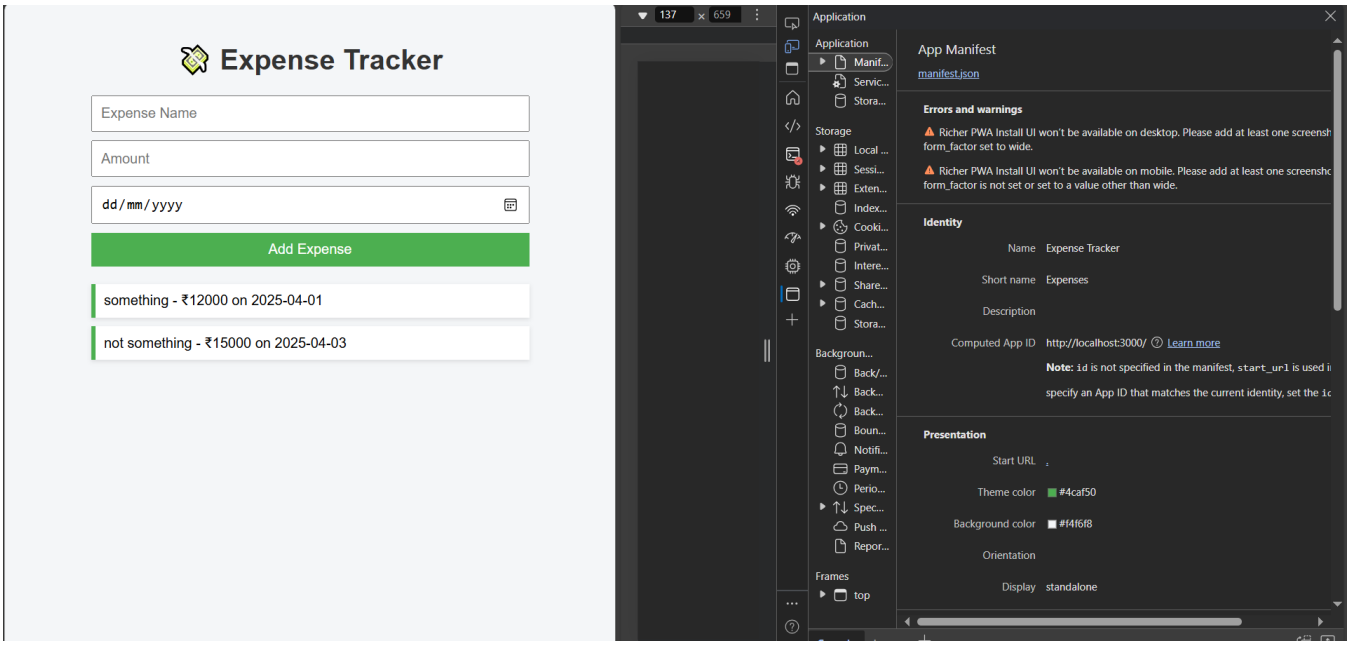
Progressive Web Apps (PWAs) are web applications that offer app-like experiences through modern web capabilities. One of the key components of a PWA is the service worker, which enables features like offline access, background sync, and push notifications. In this document, we will explore how to implement service worker events such as `fetch`, `sync`, and `push` in the context of an e-commerce application. Below is a sample implementation.

---

1. Caching Static Assets Using Install Event

The `install` event is triggered when the service worker is installed. During this phase, essential files are cached to enable offline access.

```
const CACHE_NAME = "campquest-v1"; const
ASSETS_TO_CACHE = [
  "/",
  "/index.html",
  "/src/main.jsx",
  "/CampQuest.svg",
  "/manifest.json",
];

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



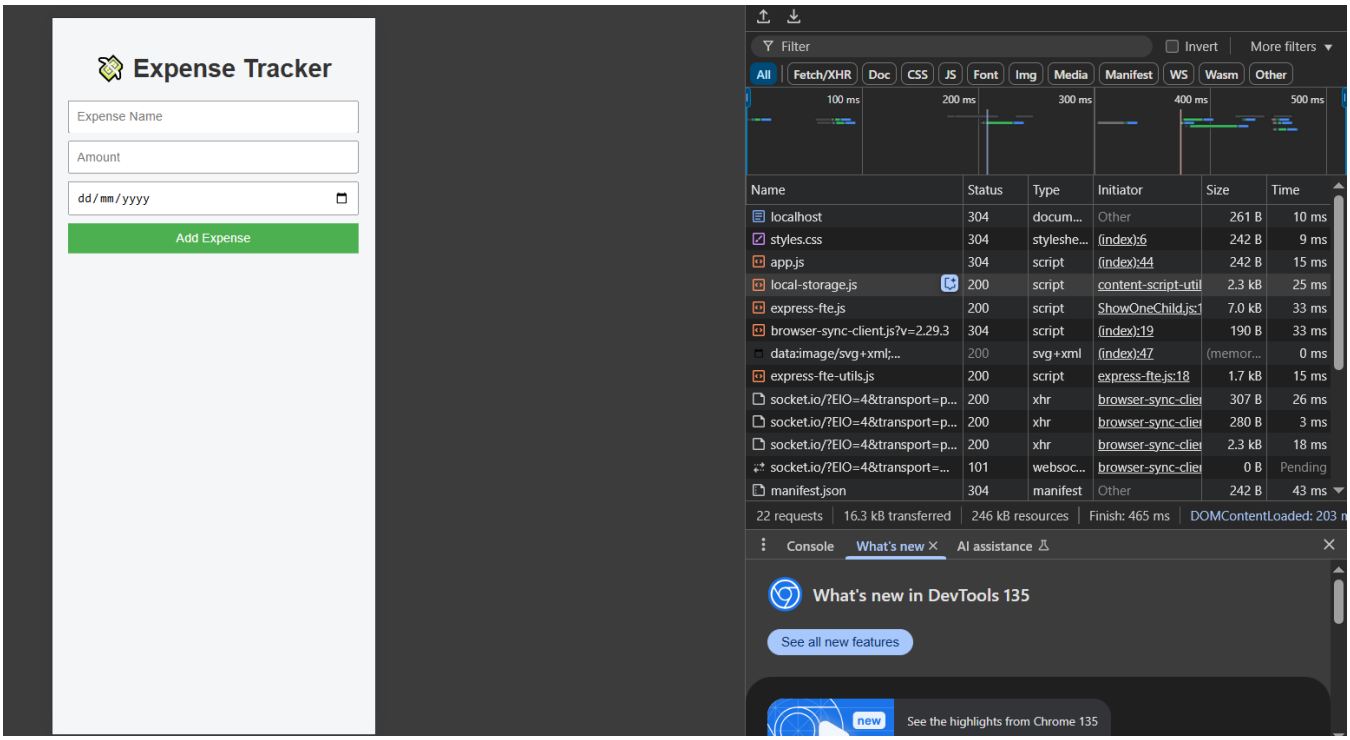
---

2. Handling Fetch Requests

The `fetch` event intercepts network requests. We use this event to implement a cache-first or network-first strategy depending on the URL path.

```
self.addEventListener("fetch", (event) => {   const url =
new URL(event.request.url);   if
(url.pathname.startsWith("/campgrounds")) {
event.respondWith(       fetch(event.request)
.then((response) => {          const responseClone =
response.clone();
caches.open(CACHE_NAME).then((cache) => {
cache.put(event.request, responseClone);
          });
          return response;
        })
        .catch(() => {          return
caches.match(event.request);
        })
    );
```

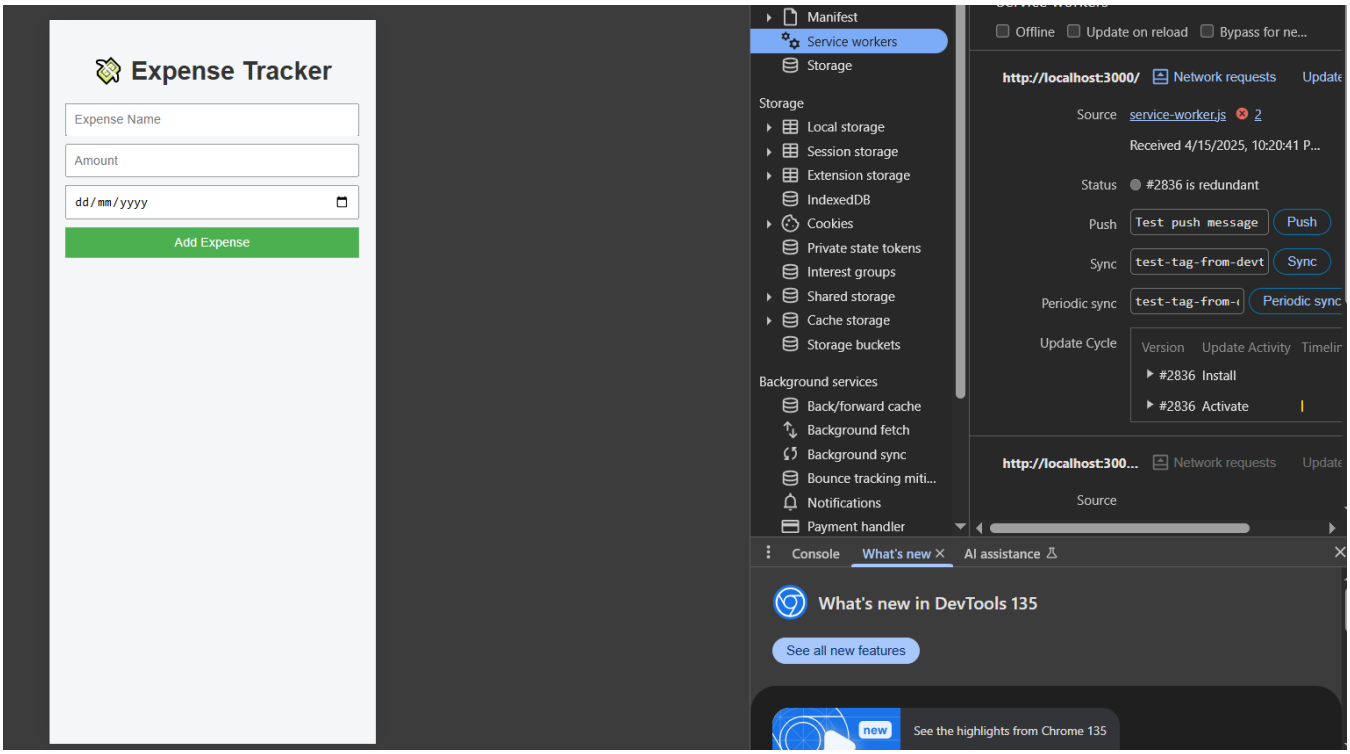
```
 } else {     event.respondWith(
caches.match(event.request).then((response) => {
return response || fetch(event.request);
    })
  );
 }
});
```



3. Background Sync (Conceptual Example)

The `sync` event is used to defer actions until the user has stable connectivity. For an e-commerce app, you could use this to sync cart data or orders.

```
self.addEventListener("sync", (event) => {
if (event.tag === "sync-cart") {
event.waitUntil(
    // Logic to sync cart data with server
  );
 }
});
```



Conclusion

Service workers are powerful tools in building resilient and engaging e-commerce PWAs. By handling `install`, `fetch`, `and sync` events effectively, you can create a seamless experience for users, even in offline or low-connectivity scenarios.

Lab 10

GitHub Pages Documentation

Introduction

GitHub Pages is a free web hosting service provided by GitHub that allows you to host static websites directly from a GitHub repository. It's perfect for portfolios, documentation, project pages, or any static content.

This guide will walk you through the process of setting up and publishing your website using GitHub Pages.

Prerequisites

- A GitHub account
- A repository containing your static website (HTML, CSS, JS)
- Basic knowledge of Git and GitHub

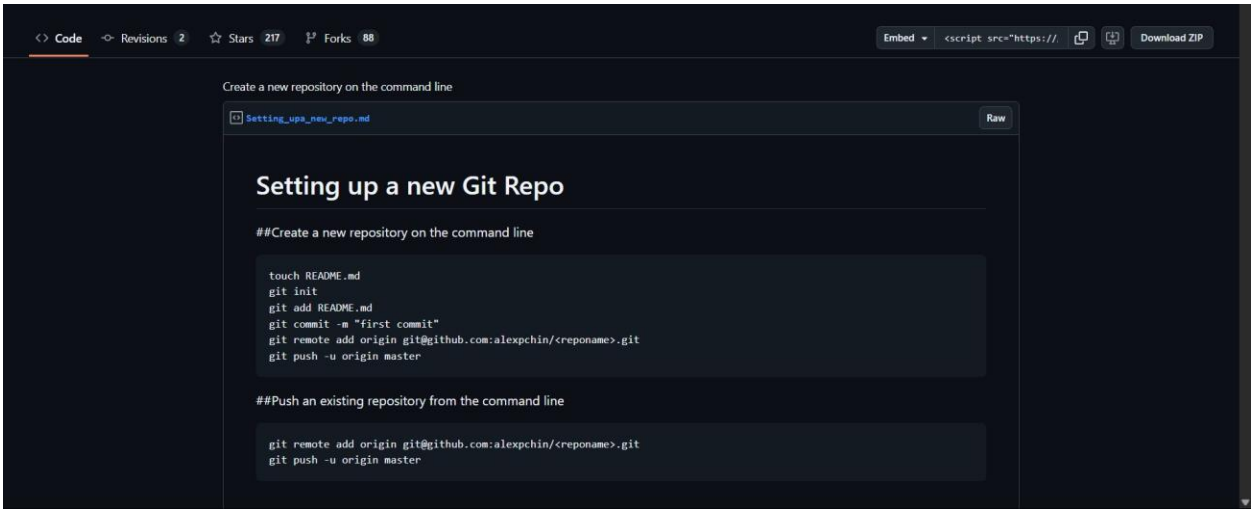
Steps to Deploy Your Website with GitHub Pages

Step 1: Create or Use an Existing Repository

If you don't have a repository yet, create one:

1. Go to github.com
2. Click on **New repository**
3. Name your repository (e.g., `my-portfolio`)

4. Initialize it with a README (optional)

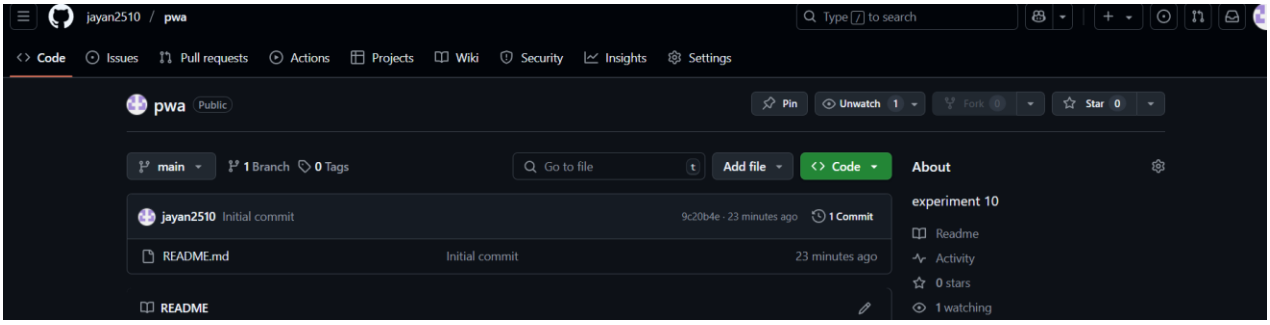


Step 2: Upload Your Website Files

Make sure your repository contains the files you want to publish, such as `index.html`.

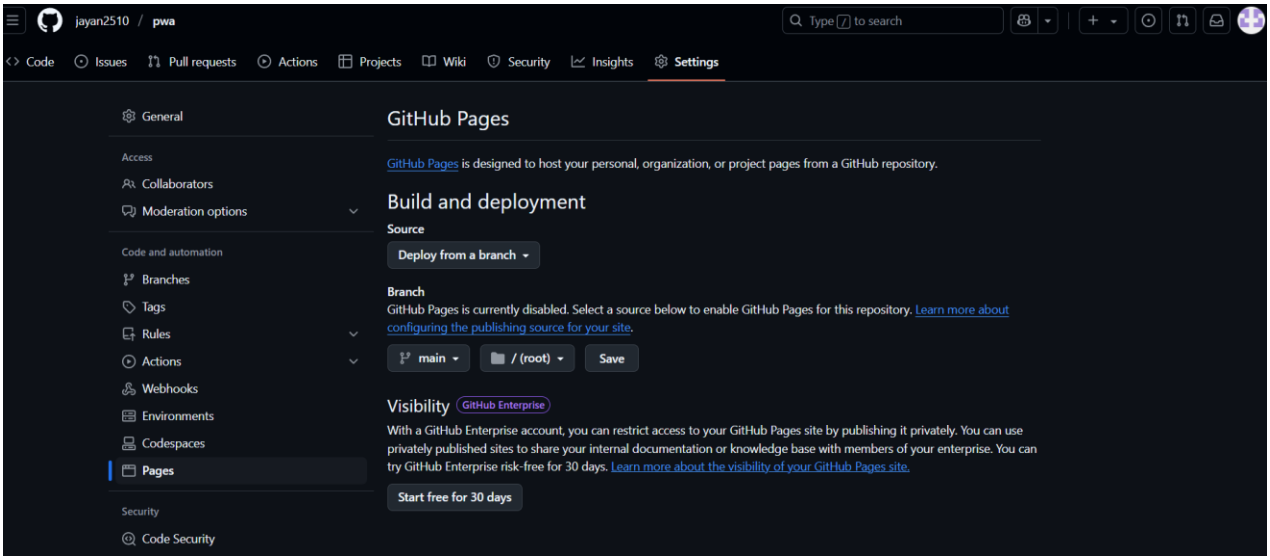
You can either:

- Drag and drop files on the web interface
- Use Git commands (`git add`, `git commit`, `git push`) from your local machine



Step 3: Enable GitHub Pages

1. Go to your repository
2. Click on the **Settings** tab
3. Scroll down to **Pages** in the left menu
4. Under **Source**, select the branch (usually `main`) and folder (e.g., `/root` or `/docs`)
5. Click **Save**

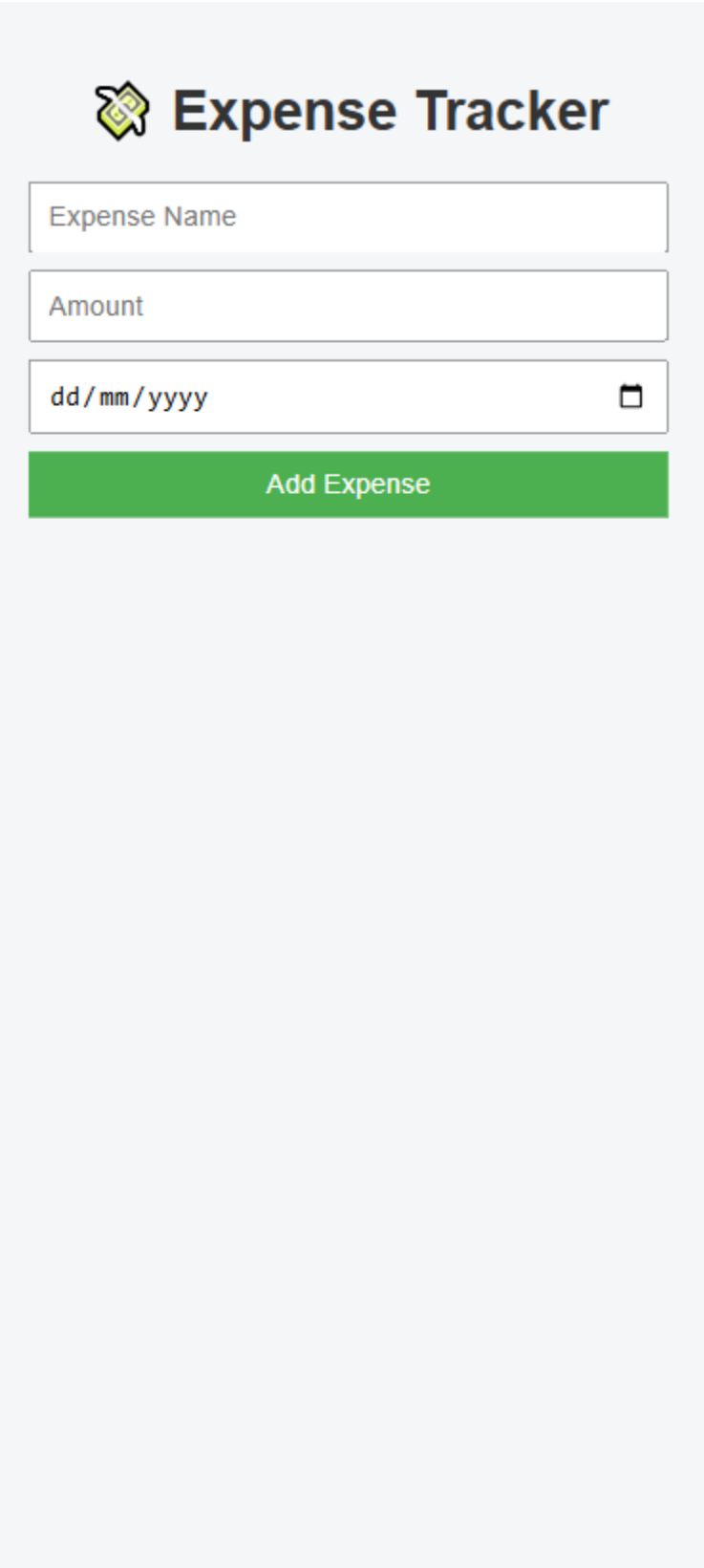


Step 4: Access Your Live Website

After enabling GitHub Pages:

- Wait a few seconds
- You'll see a link like: https://jayan2510.github.io/github_pages/

Click it to view your live site


Expense Tracker
Expense Name
Amount
dd/mm/yyyy
Add Expense

Conclusion

GitHub Pages provides an easy, fast, and free way to publish static websites directly from your GitHub repo. Whether you're showcasing a project or building a personal site, it's a powerful tool in your web development journey.

Lab 11

Google Lighthouse Extension Documentation

What is Lighthouse?

Lighthouse is an open-source tool developed by Google to audit web pages for performance, accessibility, SEO, best practices, and Progressive Web App (PWA) compatibility. It helps developers improve the quality of their websites.

Lighthouse can be accessed:

- As a **Chrome DevTools tab** (built-in)
- As a **Chrome extension**
- From the command line (`lighthouse`)
- In **PageSpeed Insights**

Why Use Lighthouse?

Lighthouse is commonly used to:

- Evaluate page **performance** (loading speed, render times)
- Improve **accessibility** (for users with disabilities)
- Follow **SEO best practices**
- Detect **common coding issues**
- Ensure PWA compliance (if applicable)

How to Use Lighthouse in Chrome Inspect Element

Step-by-step Guide

1. Open Your Website in Chrome

Go to the webpage you want to audit.

📸 **Insert Screenshot of your site opened in Chrome**

---

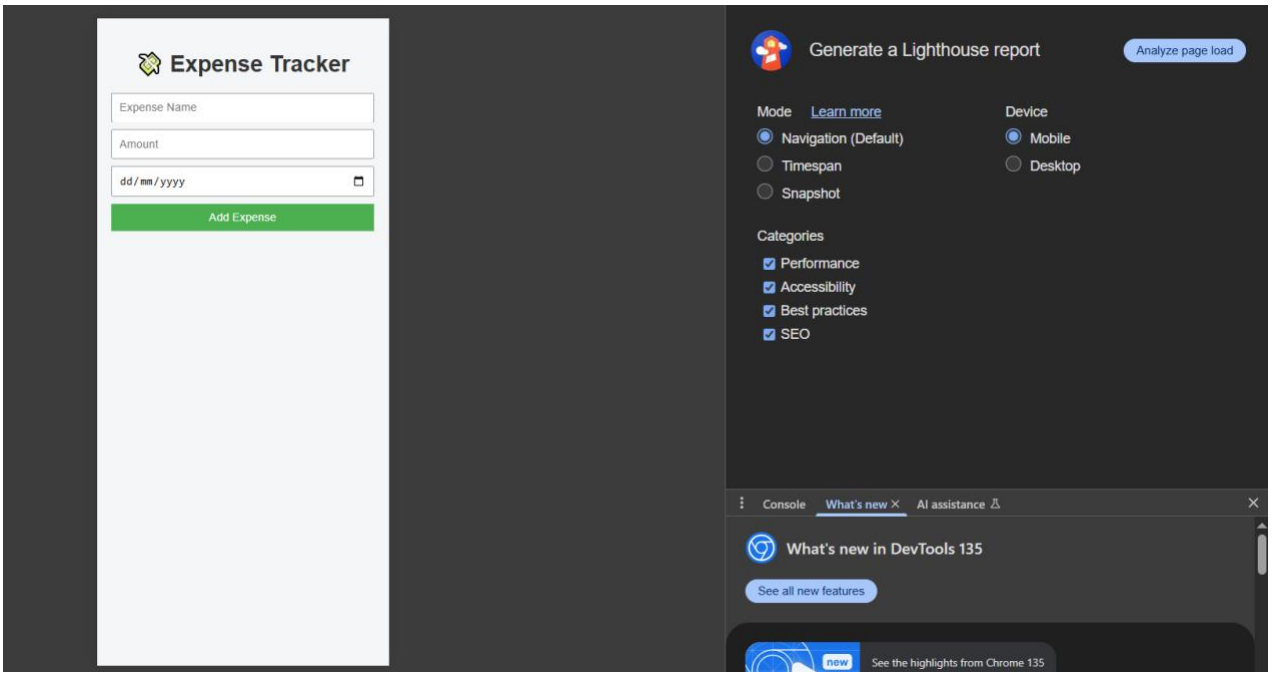
2. Open Chrome DevTools

- Right-click anywhere on the page → click **Inspect**,
  OR
- Press `Ctrl + Shift + I` (Windows) or `Cmd + Option + I` (Mac)

---

3. Configure Lighthouse Audit

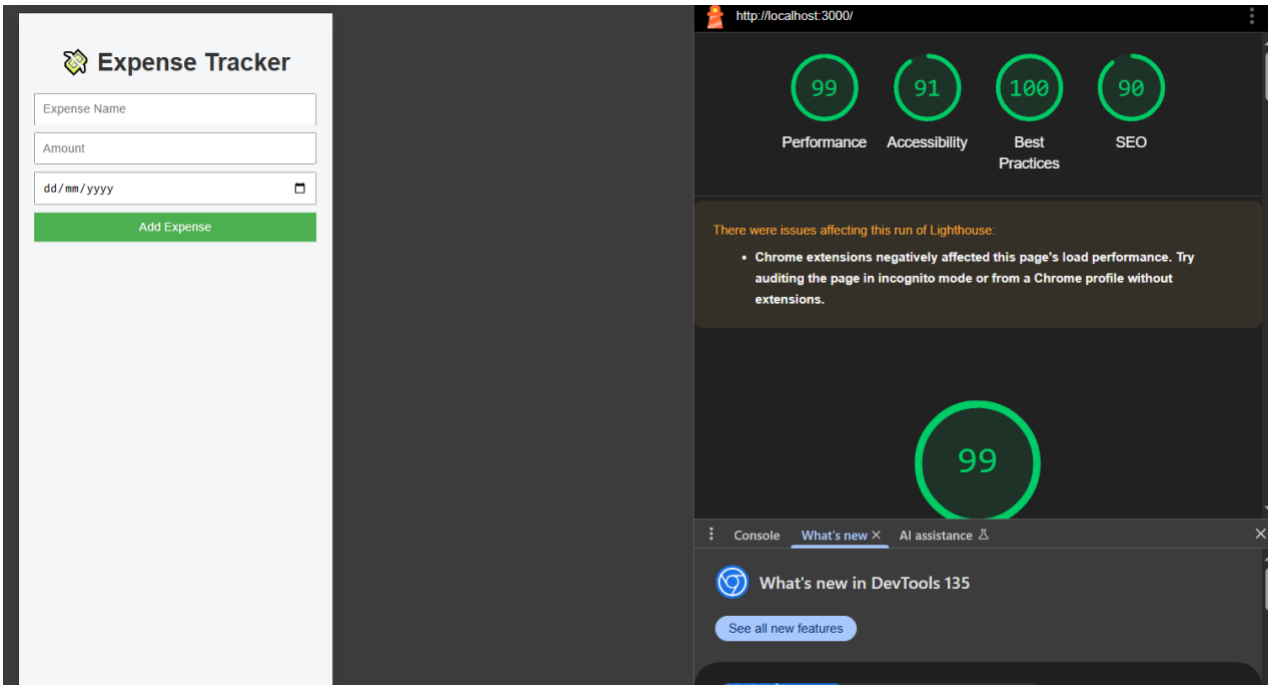
- Choose device type: **Mobile** or **Desktop**
- Select categories: Performance, Accessibility, Best Practices, SEO, PWA
- Click **Analyze page load**



---

4. View the Report

Lighthouse will run its audit and generate a score report in a few seconds.



Understanding the Scores

Metric	Description
Performance	Measures speed, loading time, and responsiveness
Accessibility	Checks usability for assistive technologies (screen readers, etc.)
Best Practices	Analyzes common coding issues, HTTPS, errors
SEO	Reviews metadata, alt tags, and other ranking factors
PWA	Tests service workers, offline mode, installability (if PWA is present)

Conclusion

Lighthouse is an essential tool for modern web developers. It provides a comprehensive report on your website's strengths and weaknesses, helping you optimize user experience and site performance with actionable insights.

Start using it regularly during development to catch issues early and ship high-quality web apps 🚀