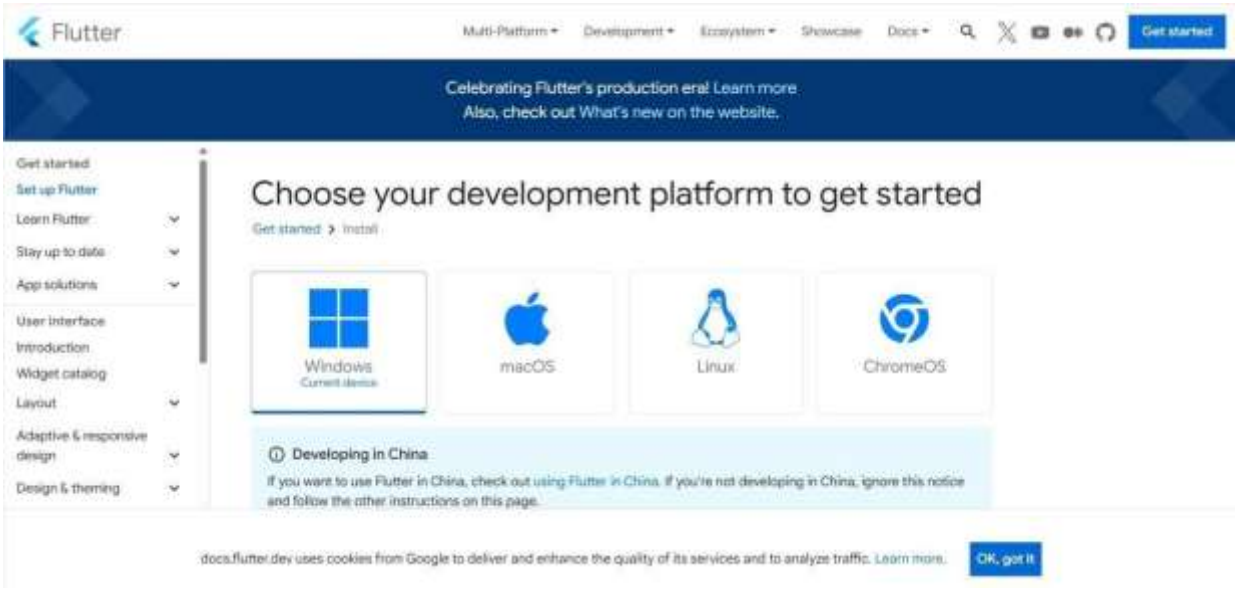
EXPERIMENT NO: - 01

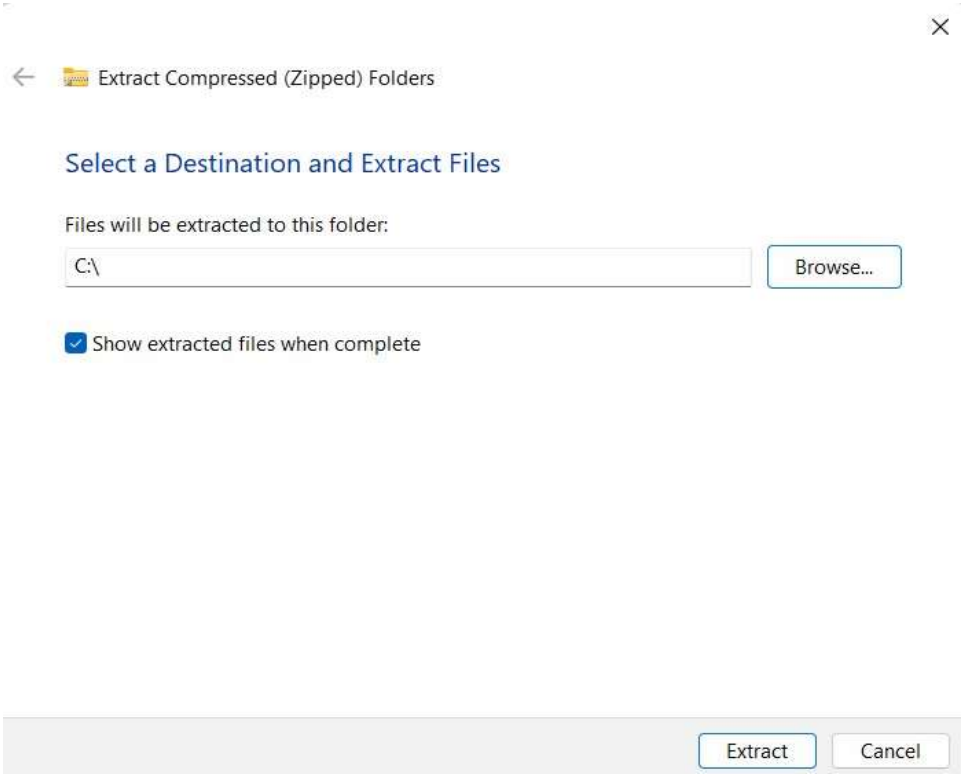
AIM: - Installation and Configuration of Flutter Environment.

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

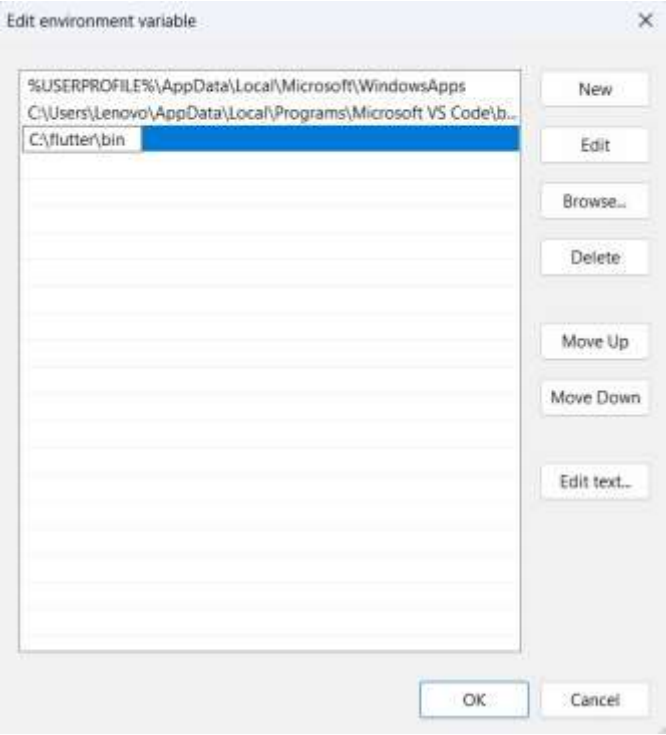

Extract Compressed (Zipped) Folders
Select a Destination and Extract Files
Files will be extracted to this folder:
C:\
Browse...
Show extracted files when complete
Extract
Cancel

Step 5 :- Add Flutter to System PATH

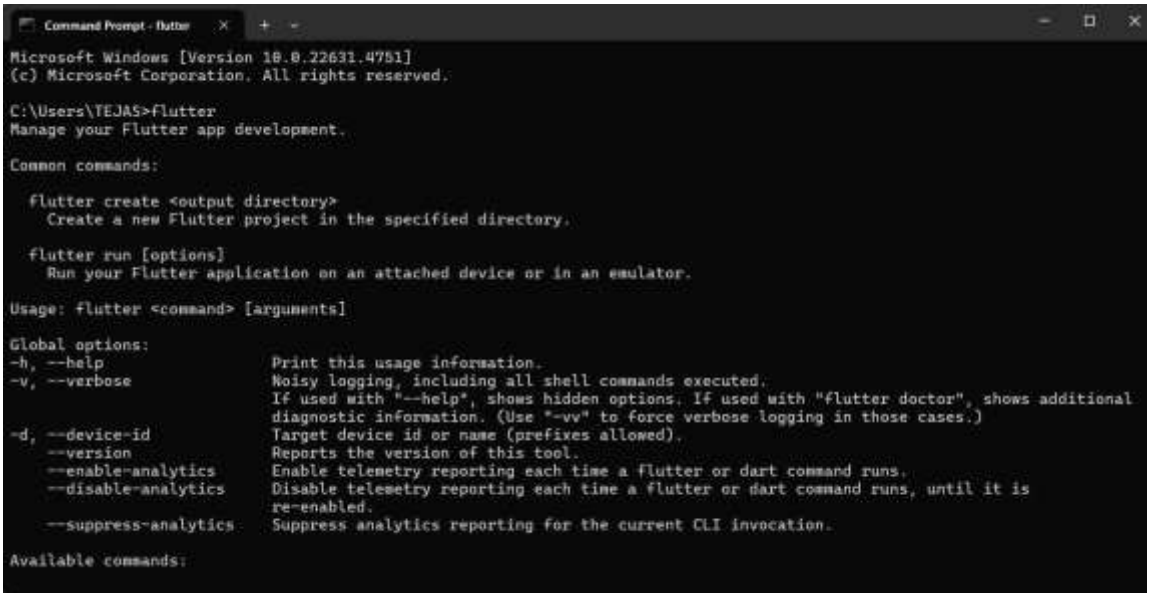
Right-click on the Start Menu > System > Advanced system settings > Environment Variables.

Under System Variables, find Path and click Edit.

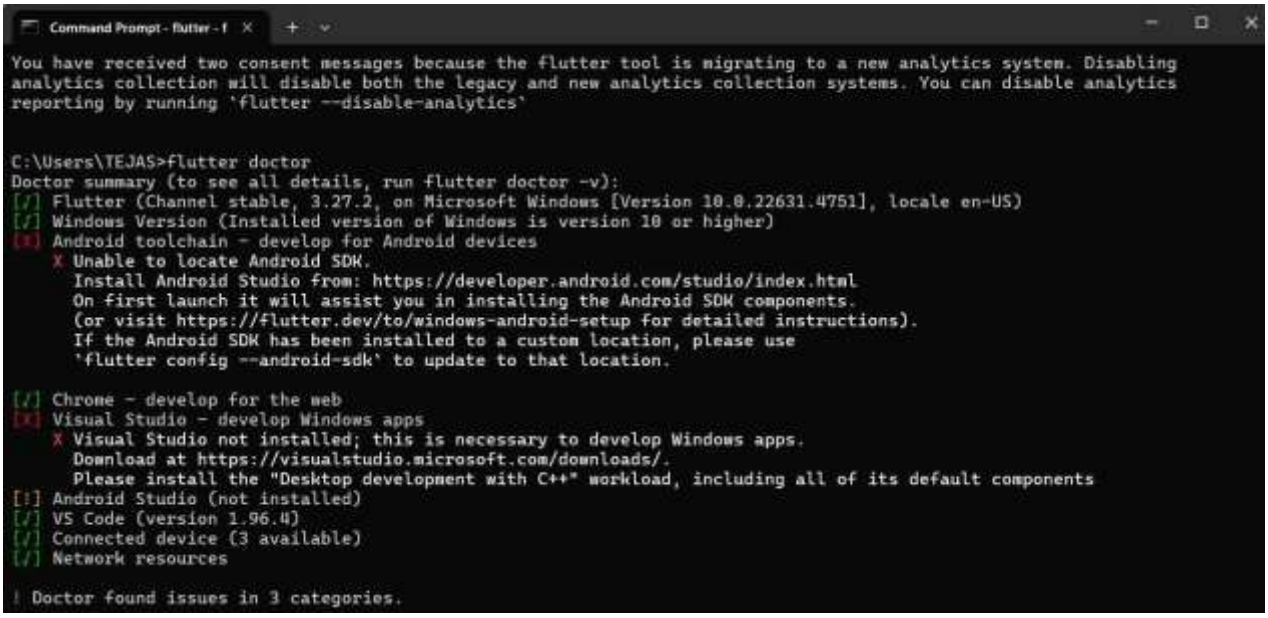
Add the full path to the flutter/bin directory (e.g., C:\flutter\bin).



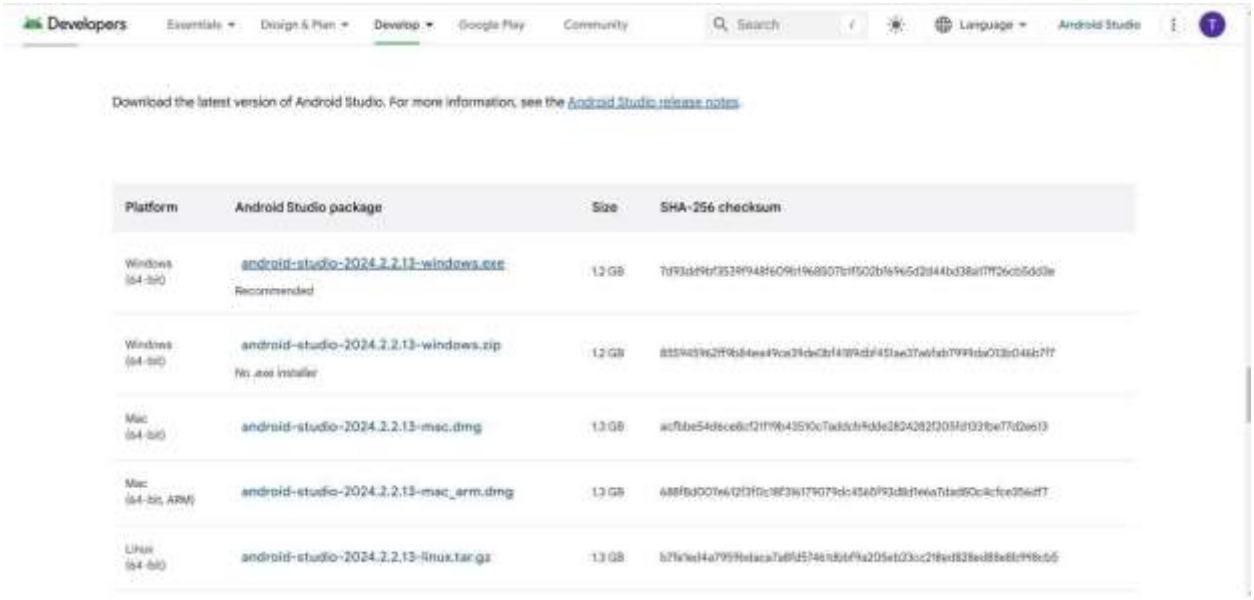
Step 6 : - Now, run the $ flutter command in command prompt.



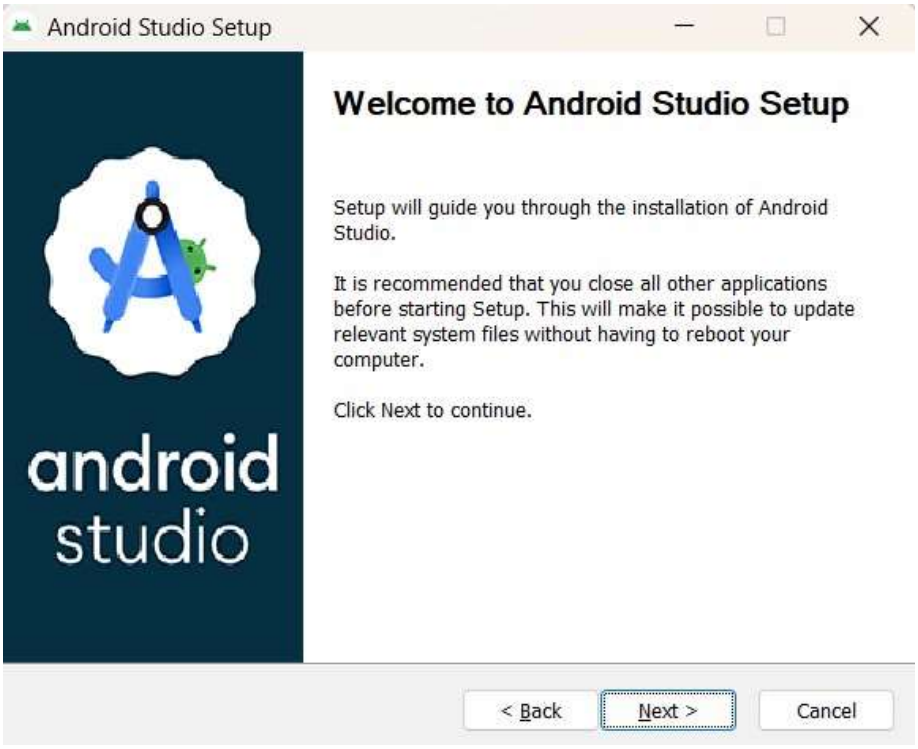
Step 7:- Run the $ flutter doctor command. This command checks for all the requirements of Flutter app development and displays a report of the status of your Flutter installation



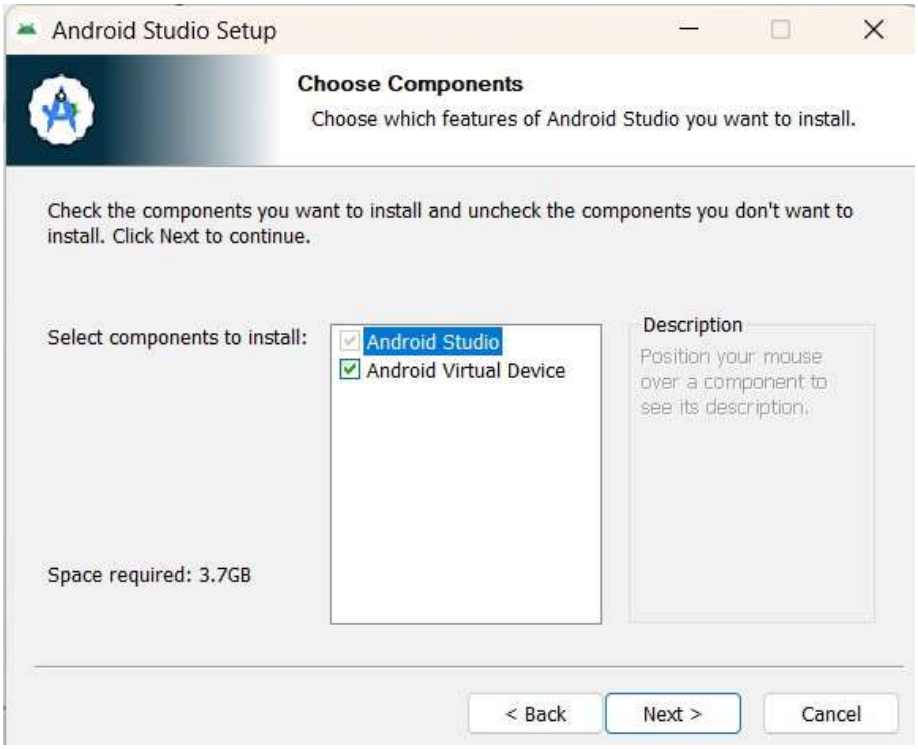
Step 8 : - Go to Android Studio and download the installer.



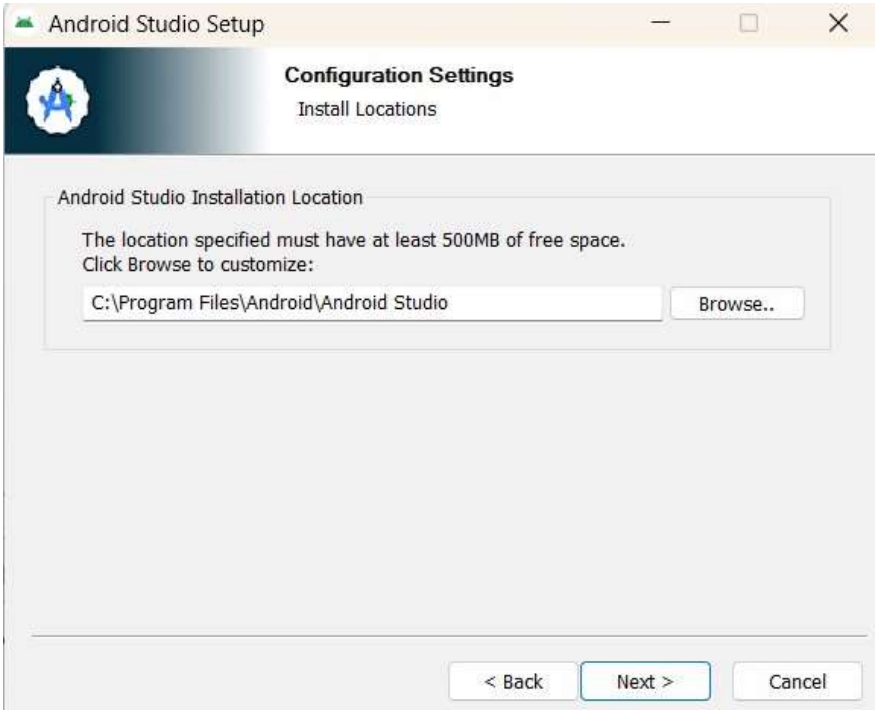
Step 8.1: - When the download is complete, open the .exe file and run it. You will get the following dialog box



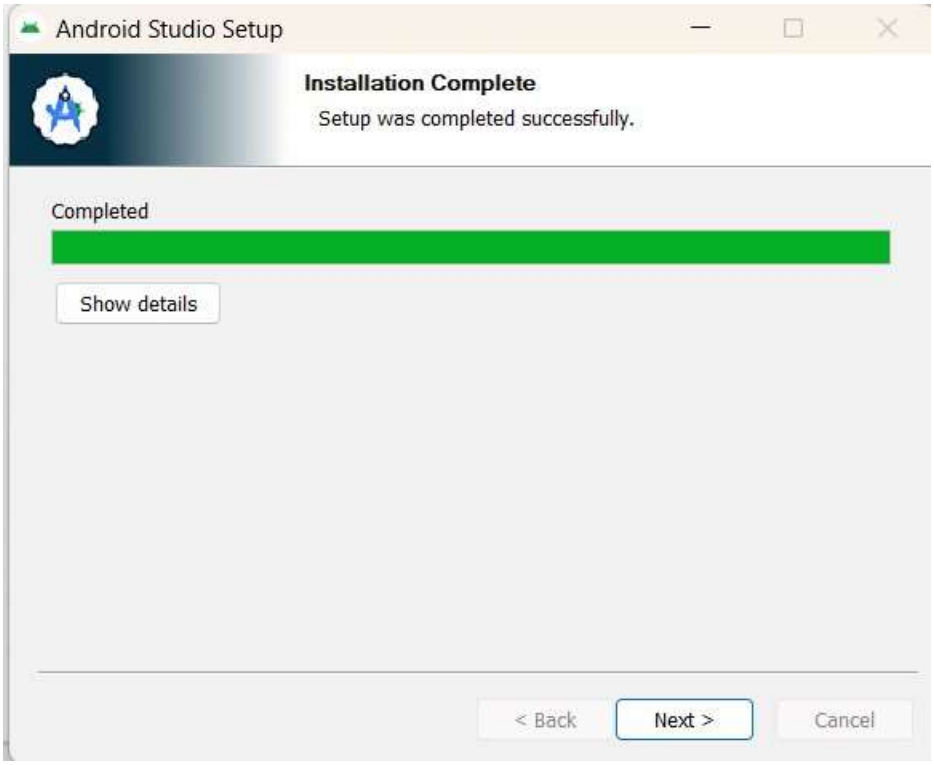
Step 8.2: - Select all the Checkboxes and Click on 'Next' Button.

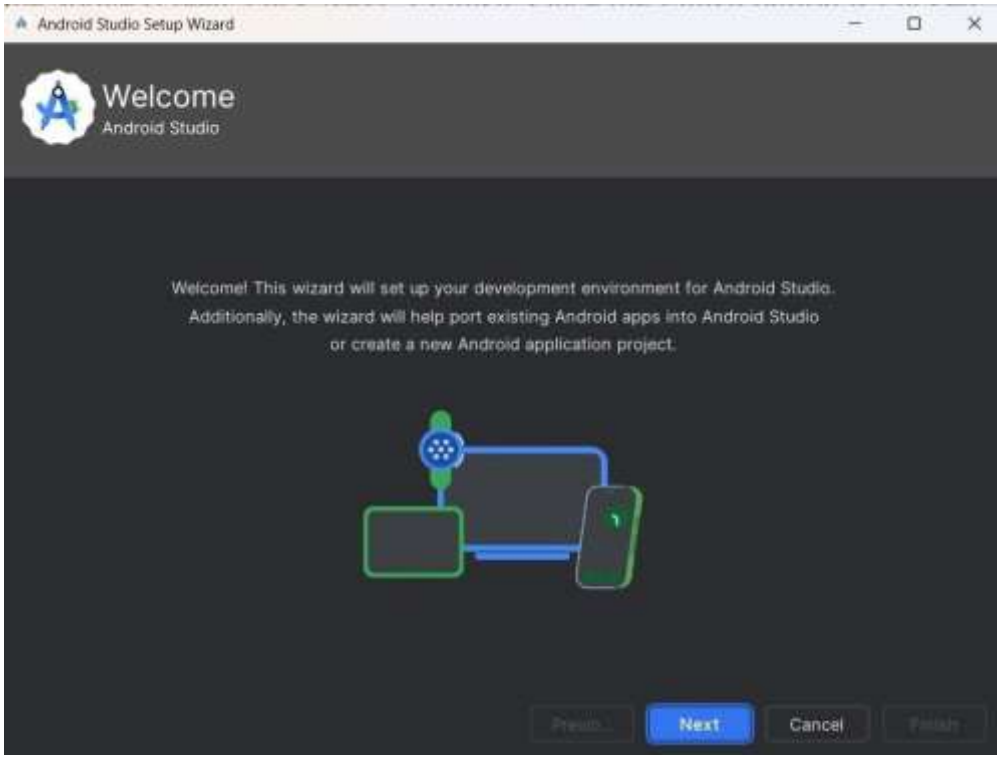


Step 8.3: - Change the destination as per your convenience and click on 'Next' Button.

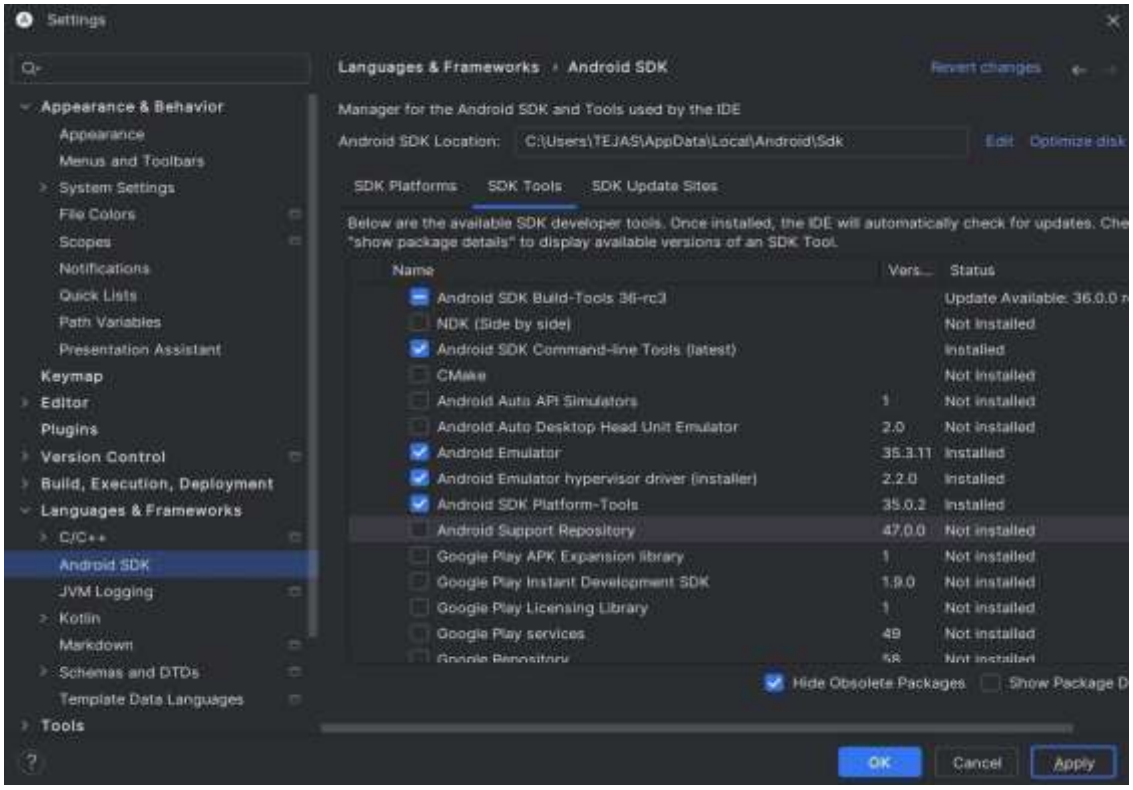


Step 8.4: - Follow the steps of the installation wizard. Once the installation wizard completes, you will get the following screen.

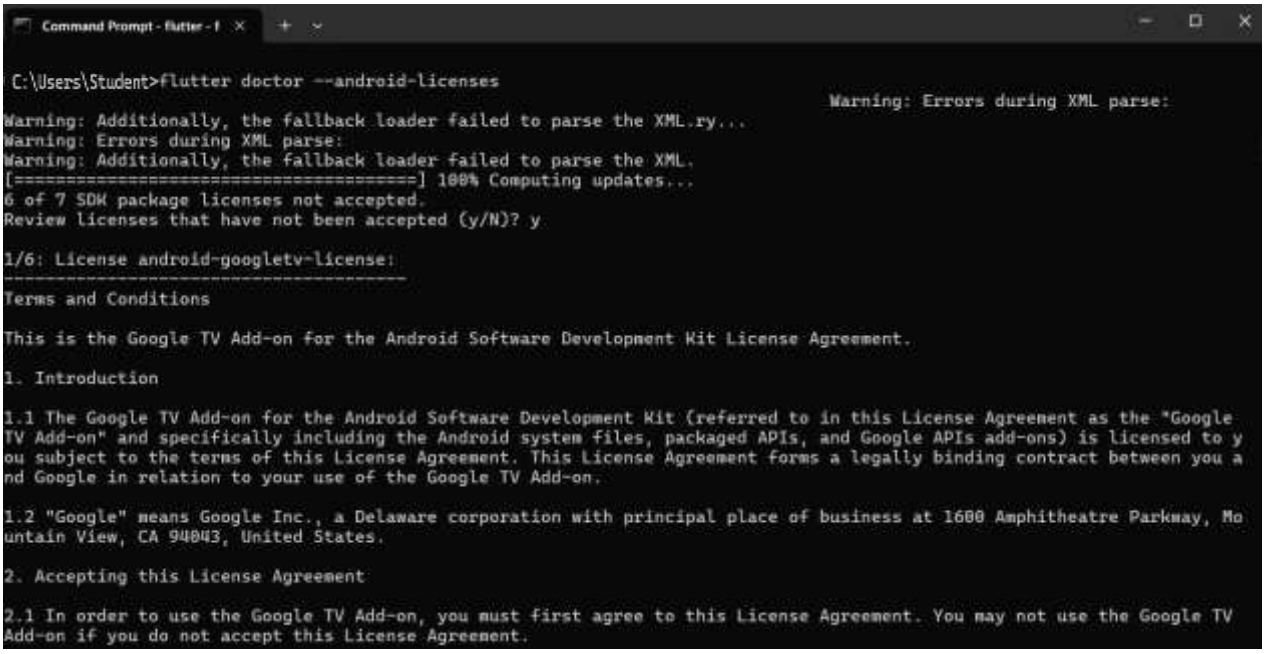


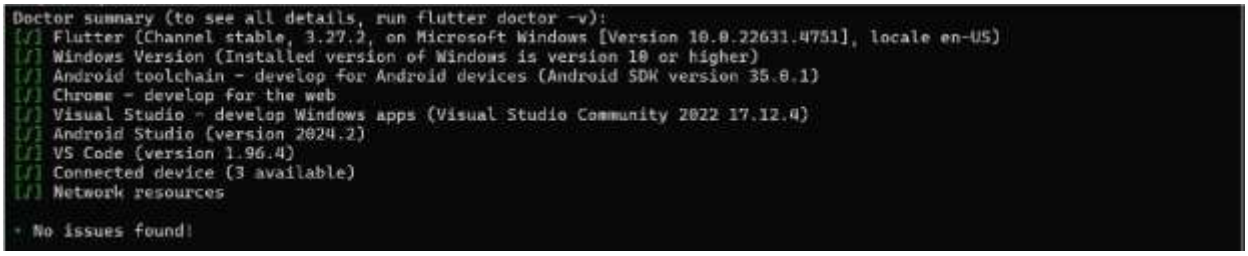


Step 8.5: - Go to Preferences > Appearance & Behavior > System Settings > Android SDK. Select the SDK Tools tab and check Android SDK Command-line Tools and Install it.

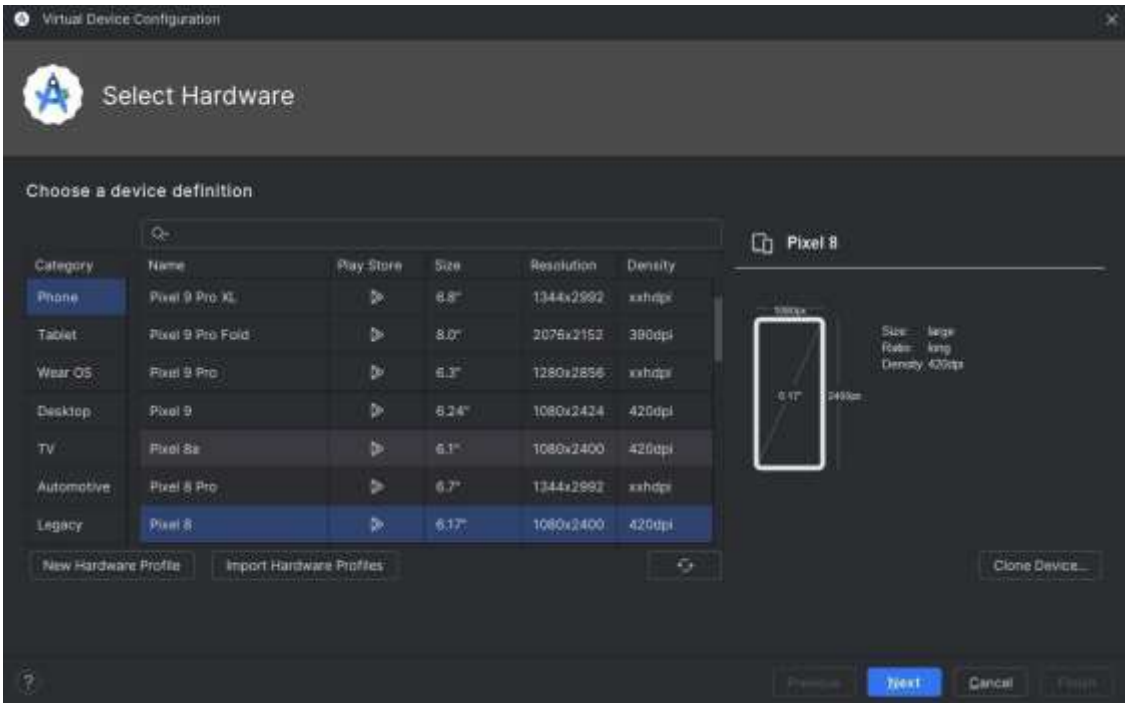


Step 9: - Open a terminal and run the following command

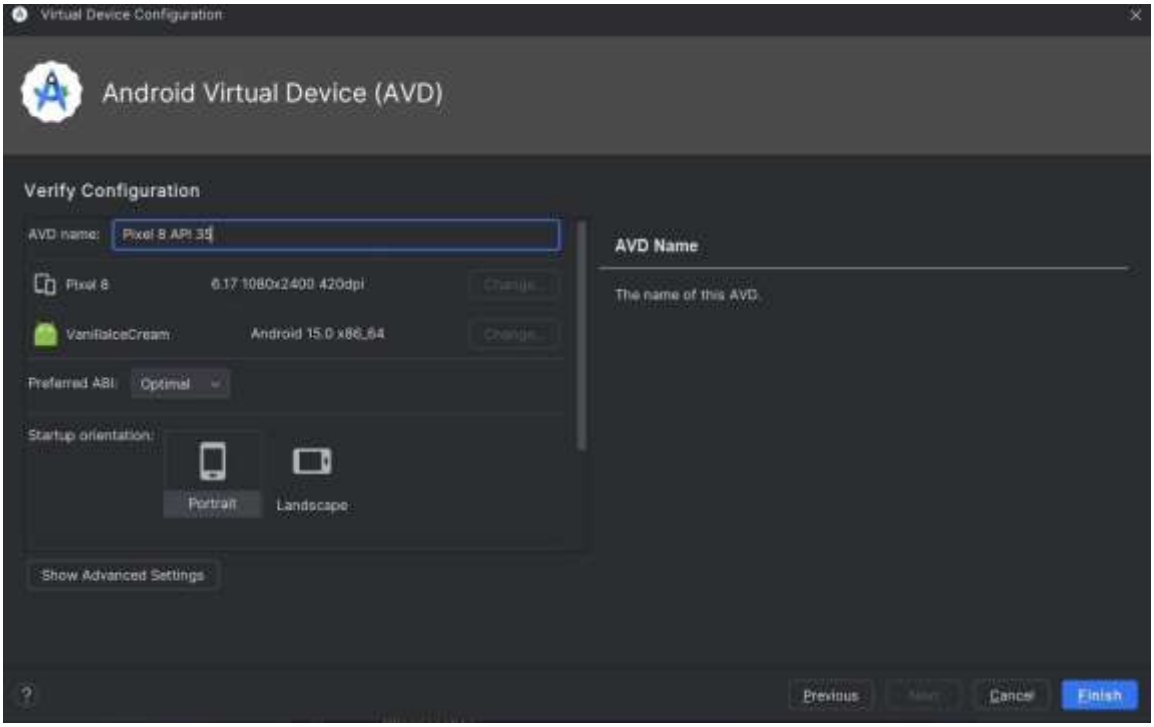




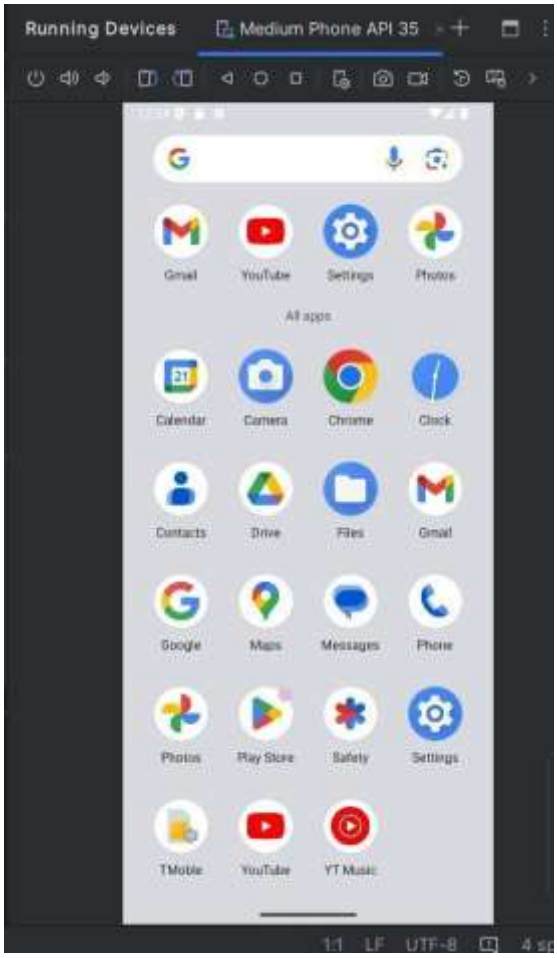
<u>Step 10: -</u> Next, you need to set up an Android emulator. It is responsible for running and testing the Flutter application



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

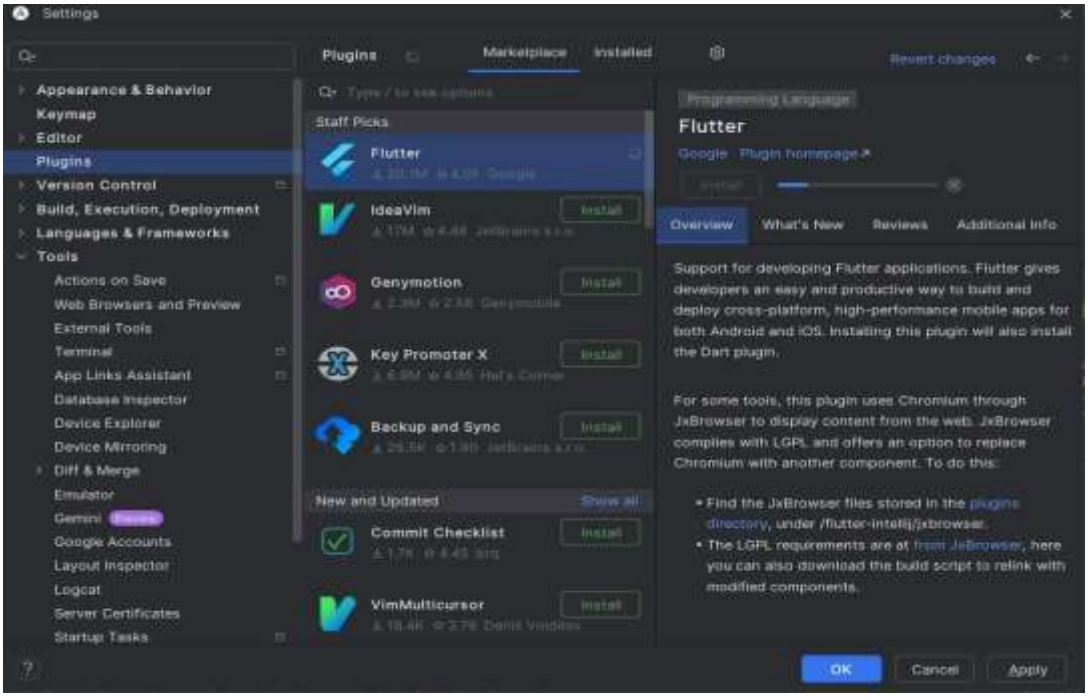


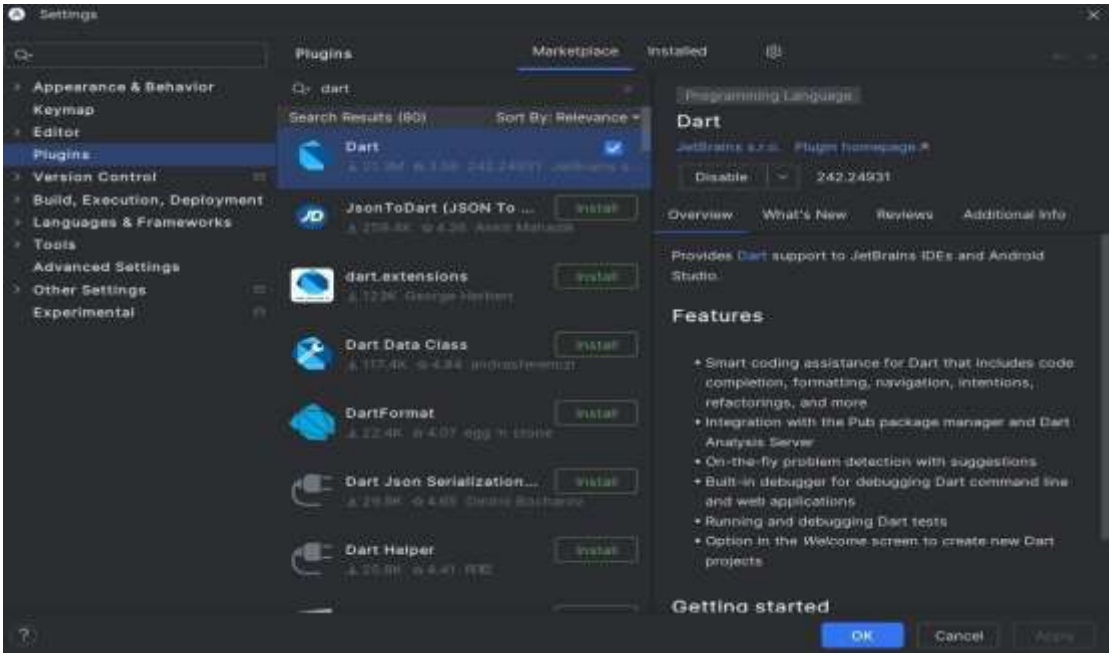
Step 10.2: - Click on the icon pointed into the red color rectangle. The Android emulator displayed as below screen



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





Conclusion – Installing Flutter, Android Studio, and Dart sets up a complete environment for cross-platform app development. Flutter provides the SDK and framework, Android Studio offers powerful tools and an emulator, while Dart serves as the primary programming language. Together, they enable smooth development, testing, and deployment of mobile apps. This setup ensures a robust and efficient workflow for creating high-performance applications.

Rushabh Jain **D15B 23**

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

import 'package:flutter/material.dart';

Theory - To design a Flutter UI, developers rely on a wide range of common widgets provided by the Flutter framework. These widgets form the building blocks of any Flutter application and are highly customizable to meet various design needs.

Widgets like Container, Row, Column, Text, Image, Icon, and ElevatedButton are frequently used to structure and display content. Layout widgets such as Padding, Align, Expanded, and Center help control spacing and positioning.

Scaffold is a powerful widget that provides a basic structure for app screens, including app bars, floating buttons, and drawers. Widgets are nested inside each other to create flexible and responsive UI designs.

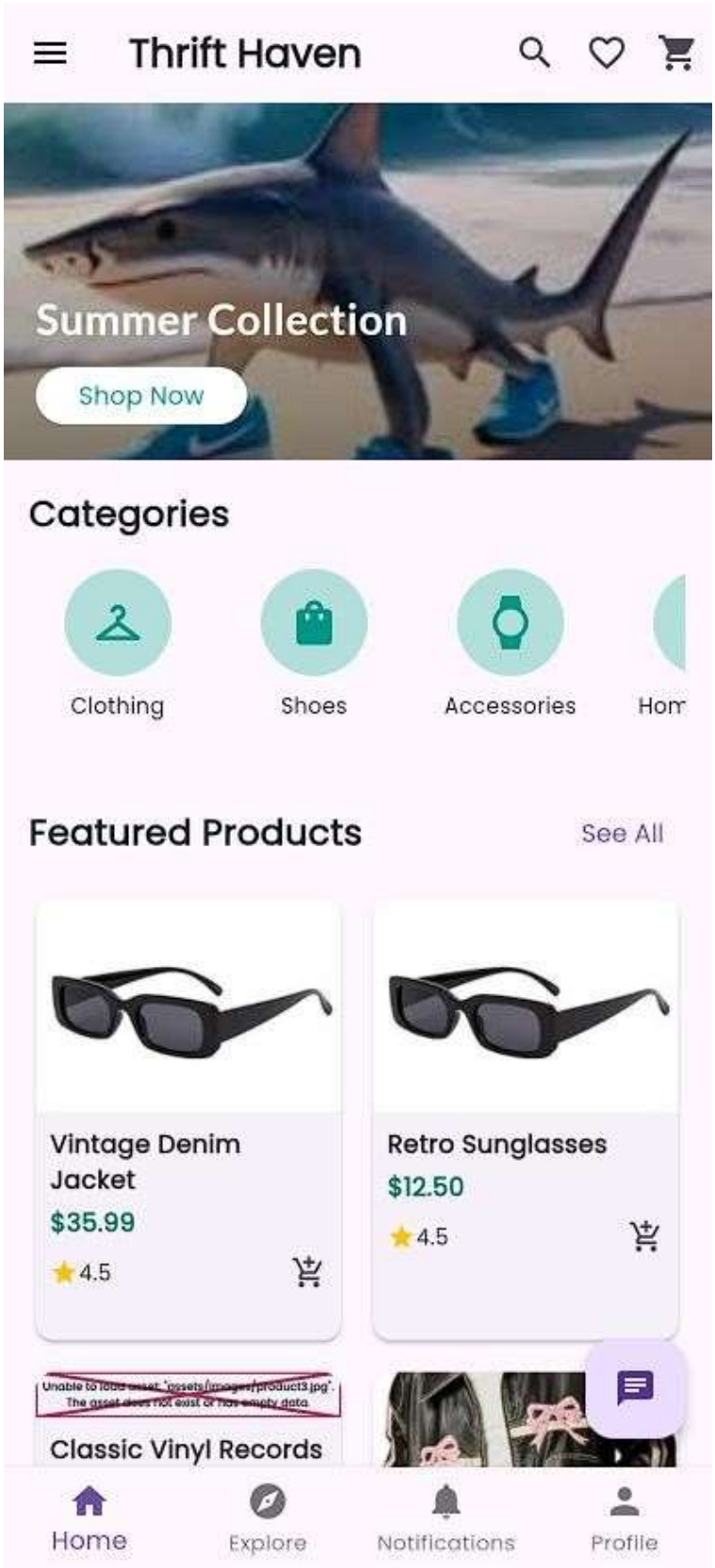
Flutter uses a declarative approach, meaning UI elements are built using code that describes their current state. Every widget is immutable and rebuilt when its state changes, ensuring smooth UI updates. This widget-based architecture makes Flutter UI development efficient, scalable, and easy to maintain.

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



Conclusion Designing UI in Flutter using common widgets offers flexibility, control, and a smooth development experience. These widgets simplify layout creation and enable responsive, beautiful interfaces. The widget tree structure promotes code reusability and organization. With built-in customization, developers can match any design requirement. Mastery of core widgets is key to building robust Flutter apps.

Aim- 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
```

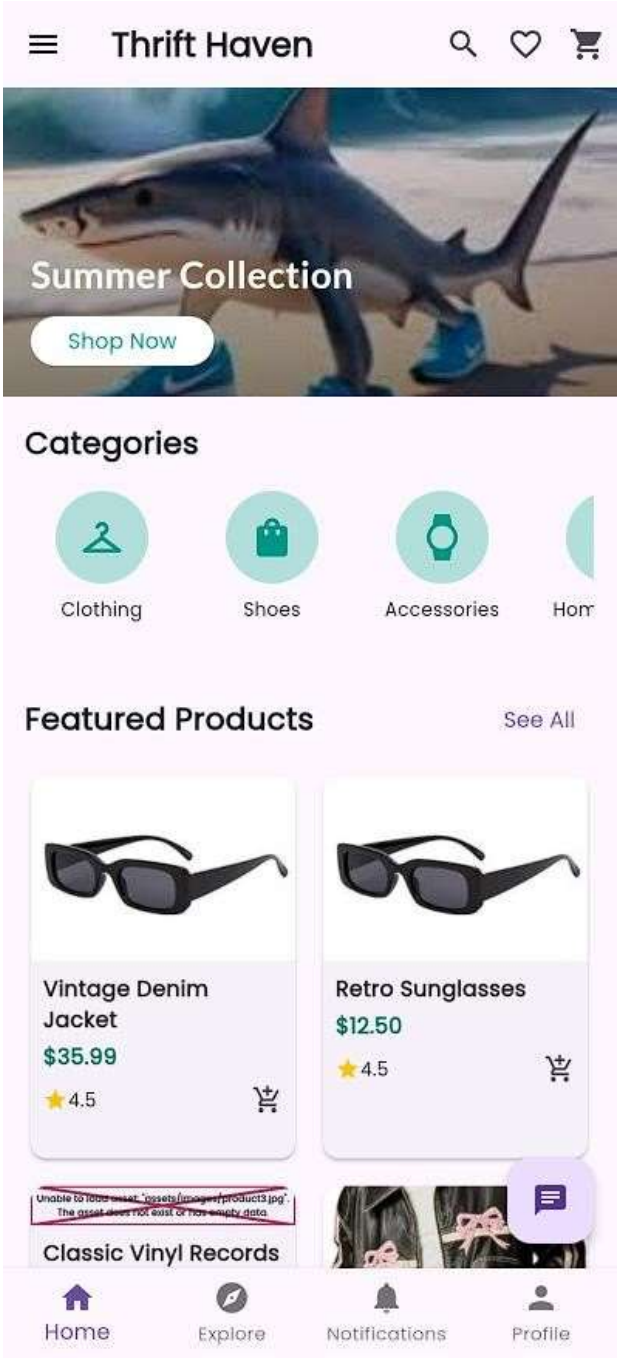
3.2 Using Custom Fonts in the App

Apply the font to text widgets:

```
Text(
```

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

Output



Conclusion- Flutter makes it easy to include images and custom fonts, enhancing both visual appeal and branding. You can load images from assets, the internet, or memory, depending on your needs. Custom fonts are added through the assets folder and configured in pubspec.yaml. This allows for consistent typography across your app. Mastering these basics helps you create polished, professional Flutter applications.

RUSHABH JAIN D15B 23

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.

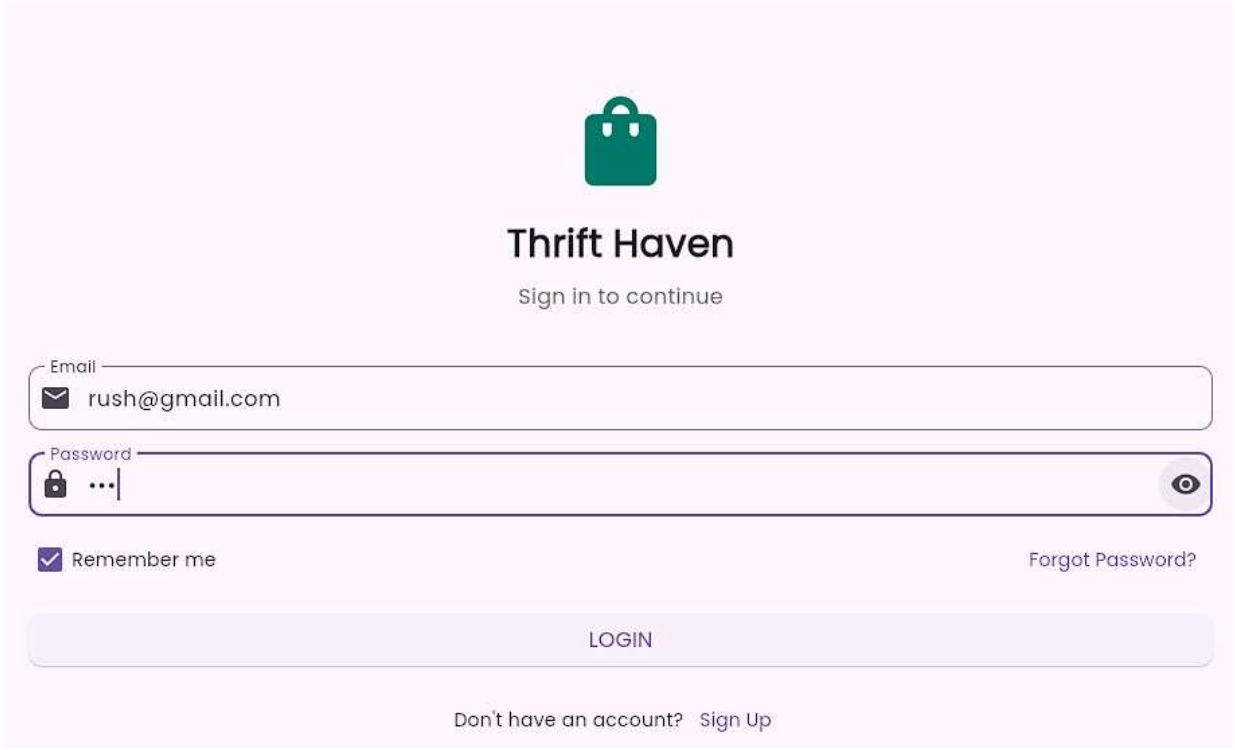
Code Implementation

```
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.

Rushabh Jain d15b 23

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.

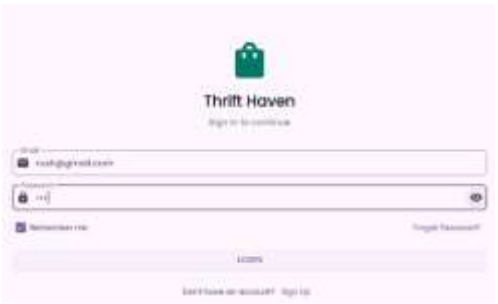
2. Gesture Detection in Flutter

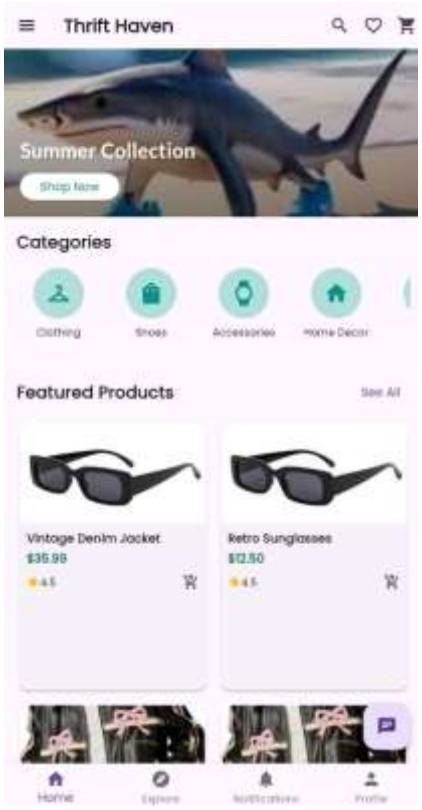
Flutter's GestureDetector widget captures various gestures like taps, double taps, long presses, and swipes.

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.




Thrift Haven
Summer Collection
Shop Now
Categories
Clothing
Shoes
Accessories
Home Decor
Featured Products
See All
Vintage Denim Jacket
$35.99
Retro Sunglasses
$12.50
Home


Product Details
Vintage Denim Jacket
$35.99
4.5 (42 reviews)
Description
This is a high-quality thrift item in excellent condition. Perfect for adding a unique touch to your style or home. This vintage piece has been carefully selected and is ready for a new home.
Details
Condition: Excellent
Brand: Vintage Collection
Material: Premium Quality
Availability: In Stock
Select Size
S
M
L
XL
Add to Cart
Buy Now

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.

Rushabh Jain D15 B 23

MAD/PWA EXP 7

//Index.html

```
<!DOCTYPE html>
<html>
<head>
<!-- Responsive -->
<meta charset="utf-8">  <meta
name="viewport"
content="width=device-width,
initial-scale=1">
<meta http-equiv="X-UA-Compatible"
content="ie=edge">
<!-- Title -->
<title>PWA Rush</title>
<!-- Meta Tags required for
Progressive Web App -->
<meta name=
"apple-mobile-web-app-status-bar"
content="#aa7700">  <meta
name="theme-color"
content="black">
<!-- Manifest File link -->  <link
rel="manifest"
href="manifest.json">
</head>
<body>
<h1 style="color: green;"> PWA 1</h1>
<p>
Hello Rush
</p>  <script>
window.addEventListener('load', () => {
registerSW();
});
// Register the Service Worker
async function registerSW() {  if
('serviceWorker' in navigator) {
try {  await navigator
.serviceworker
.register('serviceworker.js');
}  catch
(e) {
console.log
('SW
```

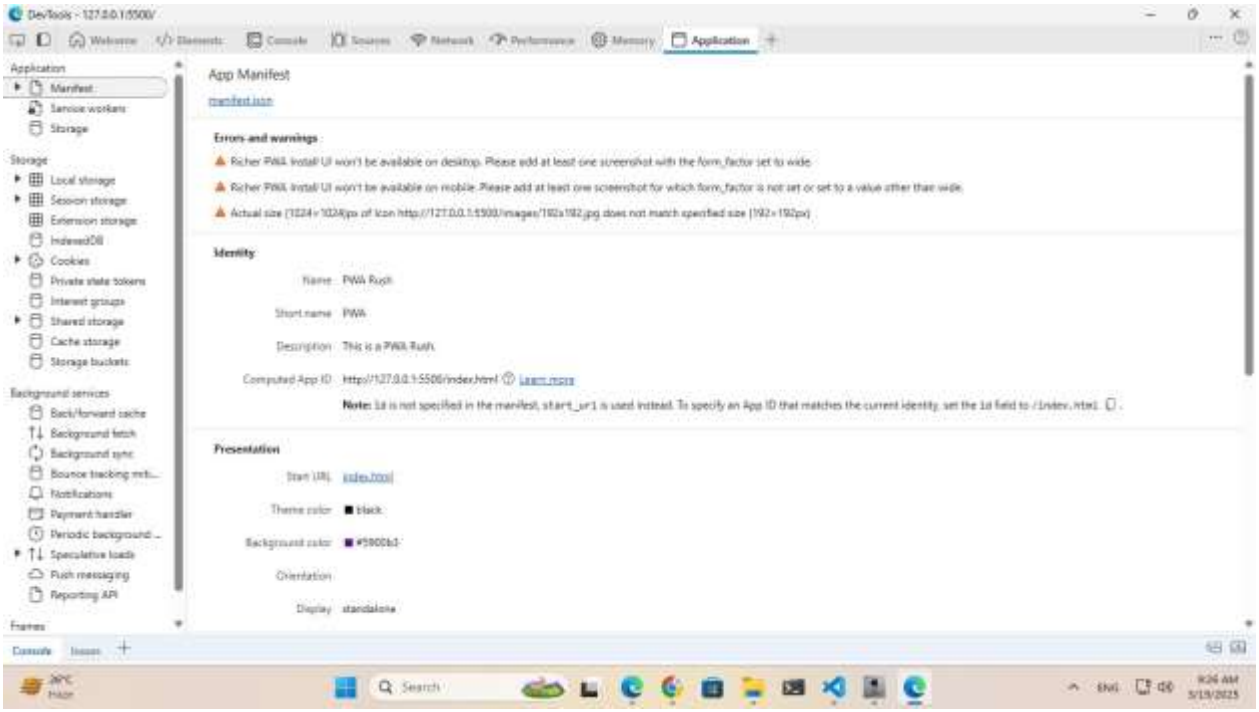
```
registratio
n failed');
}
}
}
</script>
</body>
</html>
```

//manifest.json

```
{
    "name":"PWA Rush",
    "short_name":"PWA",
    "start_url":"index.html",
    "display":"standalone",
    "background_color":"#5900b3",
    "theme_color":"black",
    "scope": ".",
    "description":"This is a PWA Rush.",
    "icons":[
    {
    "src":"images/192x192.jpg",
    "sizes":"192x192",
    "type":"image/png"
    }
    ]
}
```

//serviceworker.js var

```
staticCacheName = "pwa";
self.addEventListener("install", function (e) {
e.waitUntil(
caches.open(staticCacheName).then(function (cache) { return cache.addAll(["/"]);
})
);  });
self.addEventListener("fetch", function (event) {
console.log(event.request.url);  event.respondWith(
caches.match(event.request).then(function (response) { return response ||
fetch(event.request);
}));

});
```



// SITE INSTALLED AS APP

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
});
```

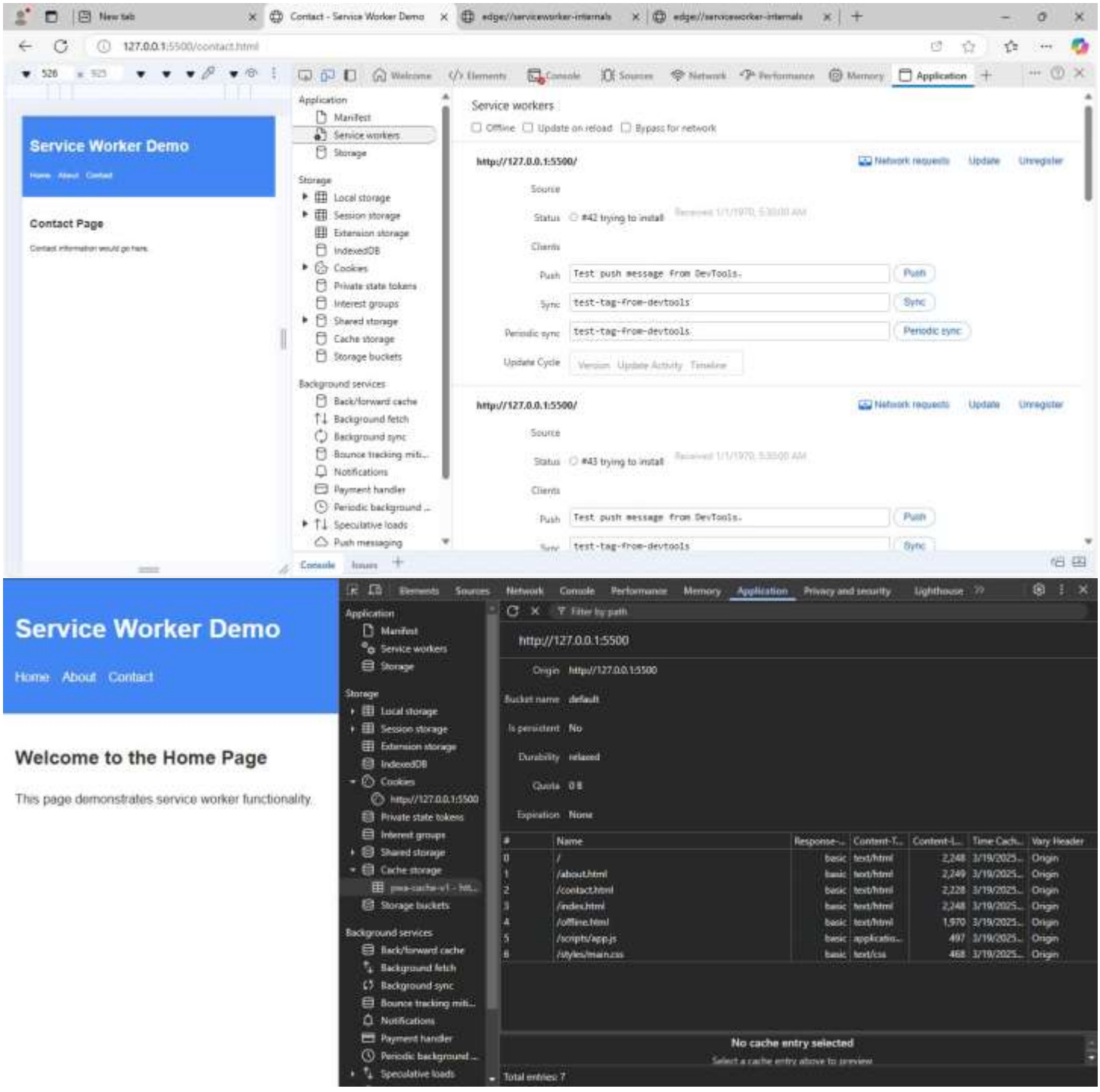
Code

```
// service-worker.js
const CACHE_NAME = 'pwa-cache-v1';

// Resources to cache during installation
const urlsToCache = [
  '/',
  '/index.html',
  '/about.html',
  '/contact.html',
  '/offline.html',
  '/styles/main.css',
  '/scripts/app.js'
];

// Install event handler self.addEventListener('install',
function(event) {   console.log('[Service Worker]
Installing...');

  // Perform install steps
  event.waitUntil(
    caches.open(CACHE_NAME)
      .then(function(cache) {
        console.log('[Service Worker] Caching app shell');
return cache.addAll(urlsToCache);
      })
      .then(function() {
        console.log('[Service Worker] Installation complete');
        return self.skipWaiting();
      })
  );
});

// Activate event handler
self.addEventListener('activate', function(event) {
  console.log('[Service Worker] Activating...');

  // Clean up old caches
event.waitUntil(
```

```
   caches.keys().then(function(cacheNames) {
return Promise.all(
       cacheNames.map(function(cacheName) {
if (cacheName !== CACHE_NAME) {
          console.log('[Service Worker] Deleting old cache:', cacheName);
return caches.delete(cacheName);
        }
       })
     );
   }).then(function() {
     console.log('[Service Worker] Activation complete');
     return self.clients.claim();
   })
  );
});

// Fetch event handler
self.addEventListener('fetch', function(event) {
  console.log('[Service Worker] Fetch event for:', event.request.url);

  event.respondWith(
caches.match(event.request)
     .then(function(response) {
       // Cache hit - return the response from the cached version
if (response) {
         console.log('[Service Worker] Returning from cache:', event.request.url);
return response;
       }

       // Not in cache - return the response from the network
       console.log('[Service Worker] Not in cache, fetching:', event.request.url);         return
fetch(event.request.clone())
         .then(function(response) {
           // Check if we received a valid response
           if (!response || response.status !== 200 || response.type !== 'basic') {
return response;
           }

           // Clone the response
           var responseToCache = response.clone();

           // Add response to cache
caches.open(CACHE_NAME)
             .then(function(cache) {
               console.log('[Service Worker] Caching new resource:', event.request.url);
```

```
            // Create explicit key-value pair
            cache.put(event.request, responseToCache);
          });

        return response;
      })
      .catch(function(error) {
        // Network request failed, try to get it from the cache
        console.log('[Service Worker] Network request failed, returning offline page');

        // For HTML requests, return the offline page
        if (event.request.headers.get('accept').includes('text/html')) {
return caches.match('/offline.html');
        }
        // You can add additional fallbacks for images, fonts, etc.
        return new Response('Network error happened', {
status: 408,
          headers: { 'Content-Type': 'text/plain' }
        });
      });
    })
  );
});

// Handle messages from the main thread
self.addEventListener('message', function(event) {
if (event.data.action === 'skipWaiting') {
self.skipWaiting();
  }
});
```

Output


Service Worker Demo
Home About Contact
Contact Page
Contact information would go here.
Application
Manifest
Service workers
Storage
Storage
Local storage
Session storage
Extension storage
IndexedDB
Cookies
Private state tokens
Interest groups
Shared storage
Cache storage
Storage buckets
Background services
Back/forward cache
Background fetch
Background sync
Notifications
Payment handler
Periodic background ...
Speculative loads
Push messaging
Service workers
Offline
Update on reload
Bypass for network
http://127.0.0.1:5500/
Network requests
Update
Unregister
Source
Status
#42 trying to install
Clients
Push
Test push message from DevTools.
Push
Sync
test-tag-from-devtools
Sync
Periodic sync
test-tag-from-devtools
Periodic sync
Update Cycle
Version
Update Activity
Timeline
http://127.0.0.1:5500/
Network requests
Update
Unregister
Source
Status
#43 trying to install
Clients
Push
Test push message from DevTools.
Push
Sync
test-tag-from-devtools
Sync

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 mit...
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

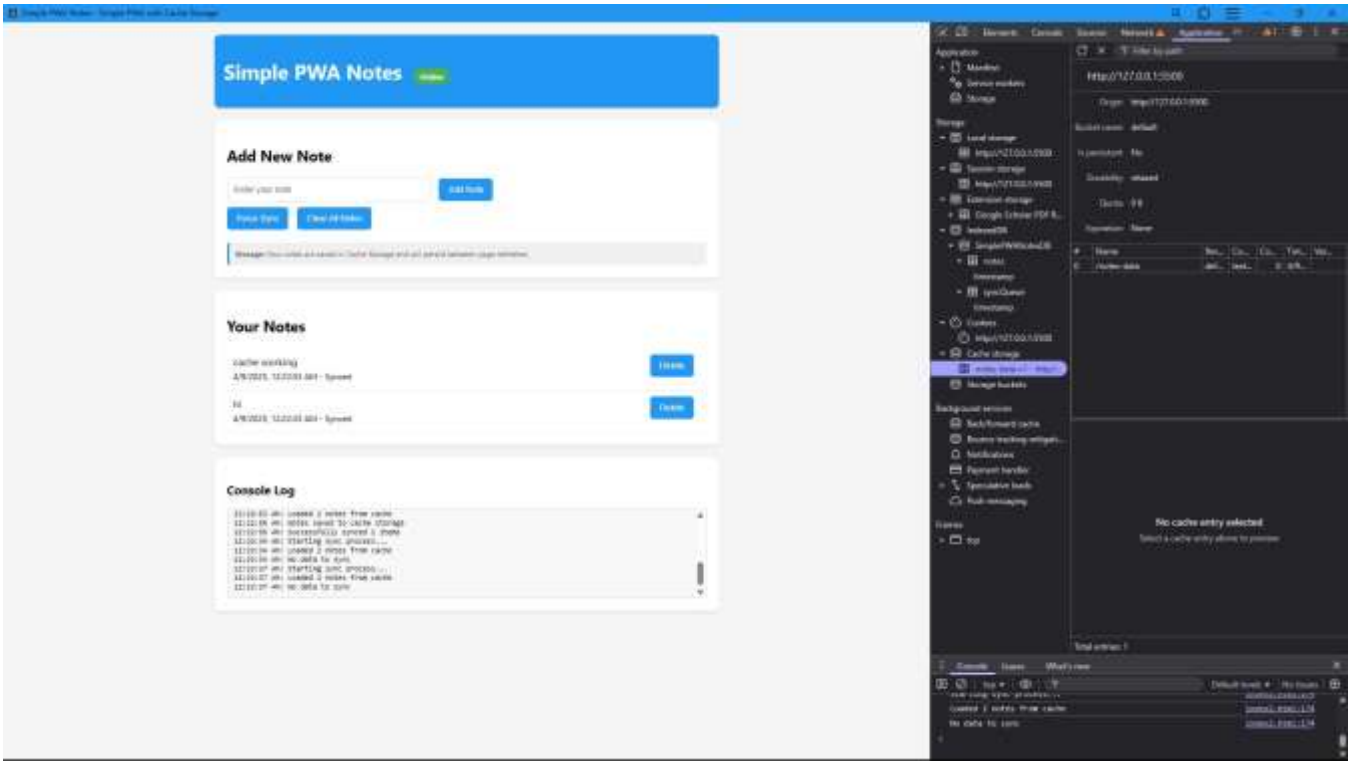
Progressive Web Apps (PWAs) are web applications that offer app-like experiences through modern web capabilities. One of the key components of a PWA is the service worker, which enables features like offline access, background sync, and push notifications. In this document, we will explore how to implement 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);
    })
  );
});
```



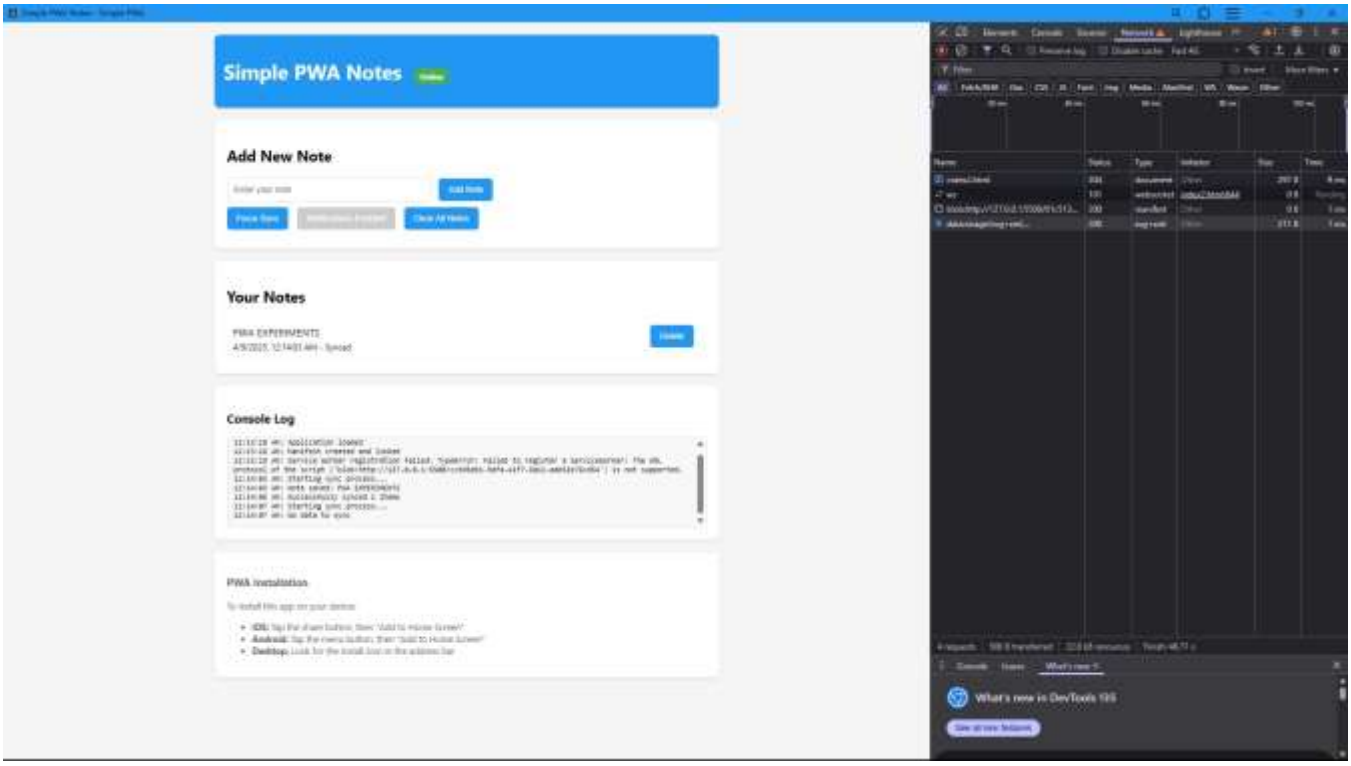
---

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 ecommerce app, you could use this to sync cart data or orders.

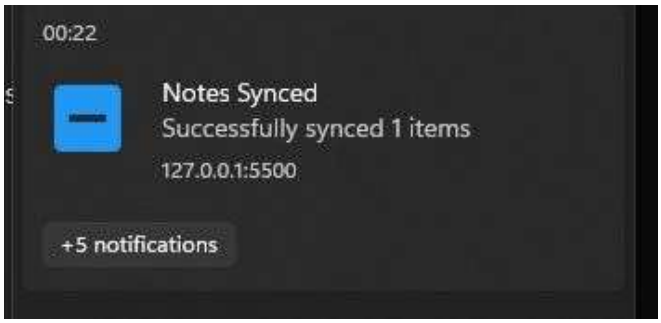
```
self.addEventListener("sync", (event) => {
if (event.tag === "sync-cart") {
event.waitUntil(
```

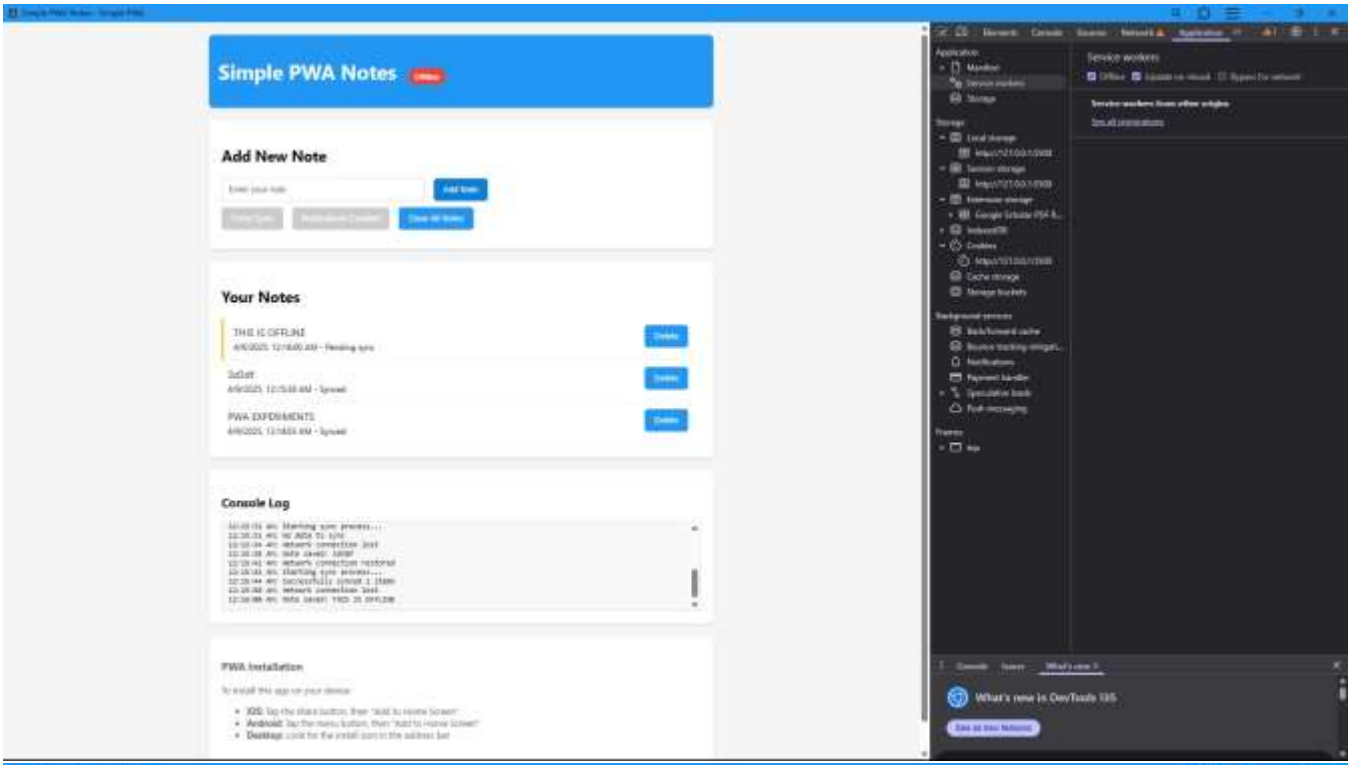
```
    // Logic to sync cart data with server
   );
  }
});
```

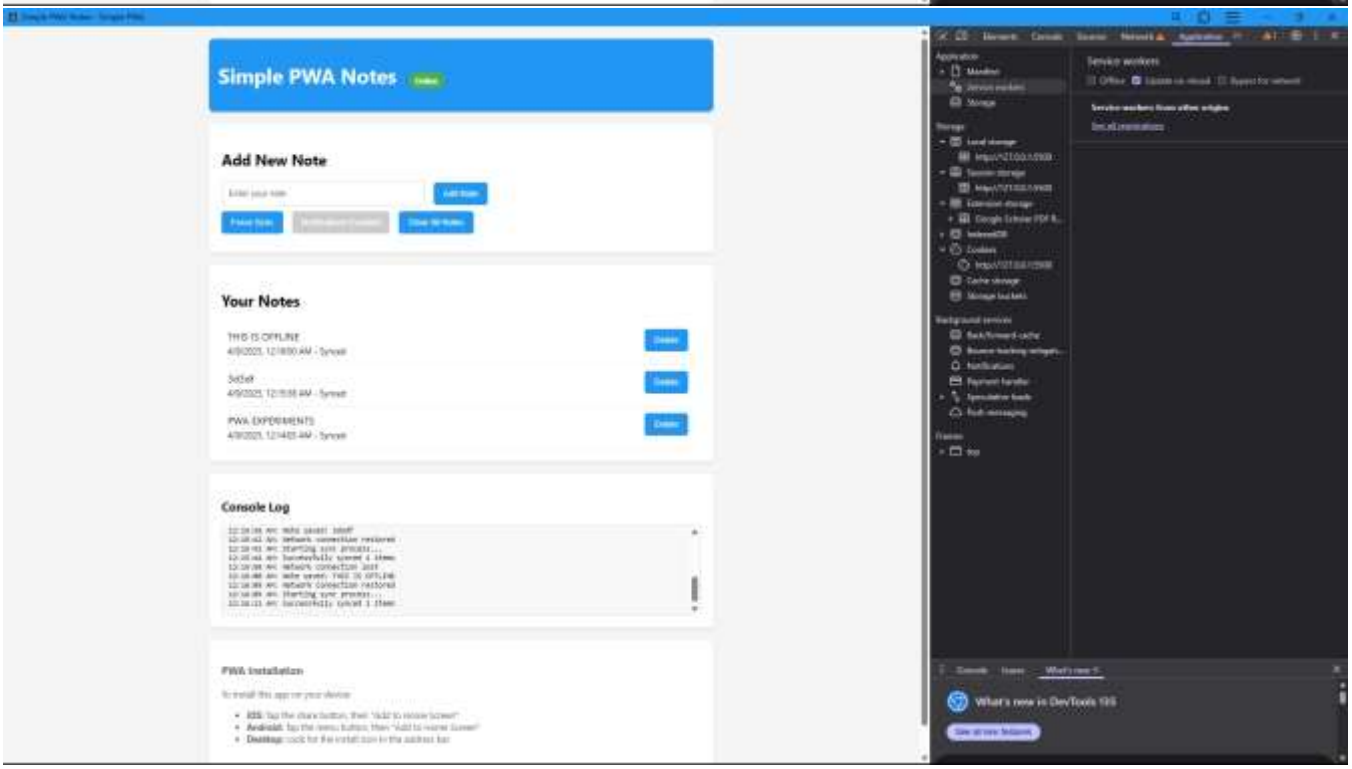
4. Push Notifications (Conceptual Example)

The `push` event is triggered when a push message is received. This could be used to notify users of new deals or order status updates.

```
self.addEventListener("push", (event) => {
const data = event.data.json();   const
options = {     body: data.body,     icon:
"/icon.png",
  };
  event.waitUntil(
self.registration.showNotification(data.title, options)
  );
});
```




Simple PWA Notes
Add New Note
Your Notes
Console Log
PWA Installation
What's new in DevTools 135


Simple PWA Notes
Add New Note
Your Notes
Console Log
PWA Installation
What's new in DevTools 135

Conclusion

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

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**

---

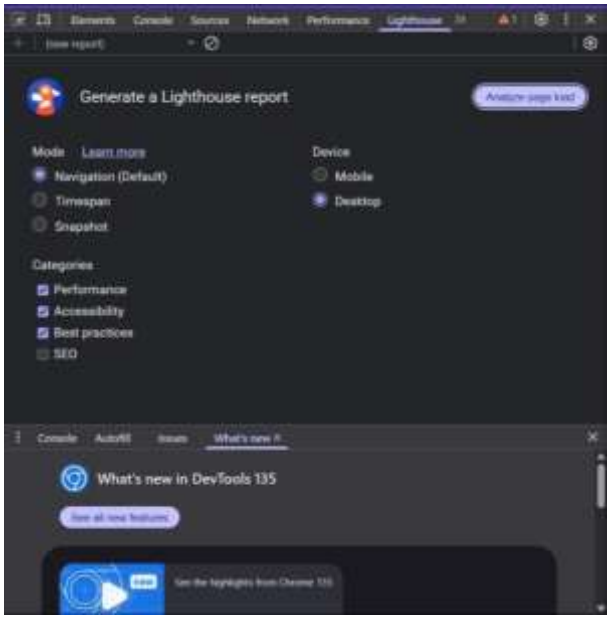
2. Open Chrome DevTools

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

---

3. Configure Lighthouse Audit

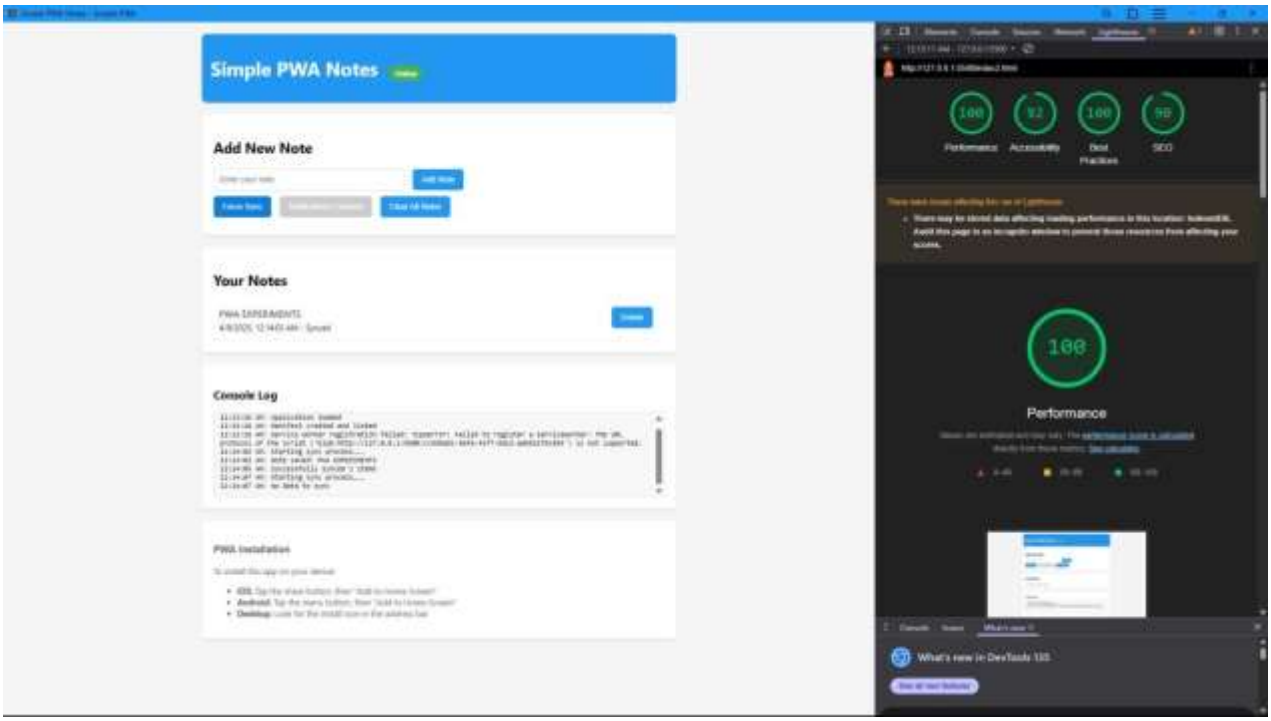
- Choose device type: **Mobile** or **Desktop**
- Select categories: Performance, Accessibility, Best Practices, SEO, PWA
- Click **Analyze page load**



---

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 🚀

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

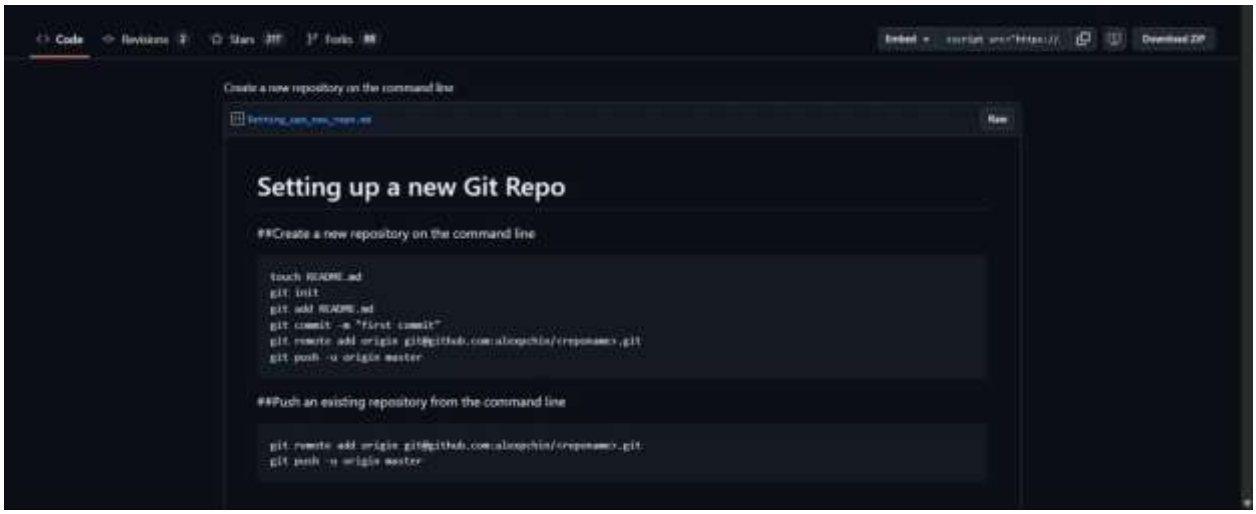
Step 1: Create or Use an Existing Repository

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)

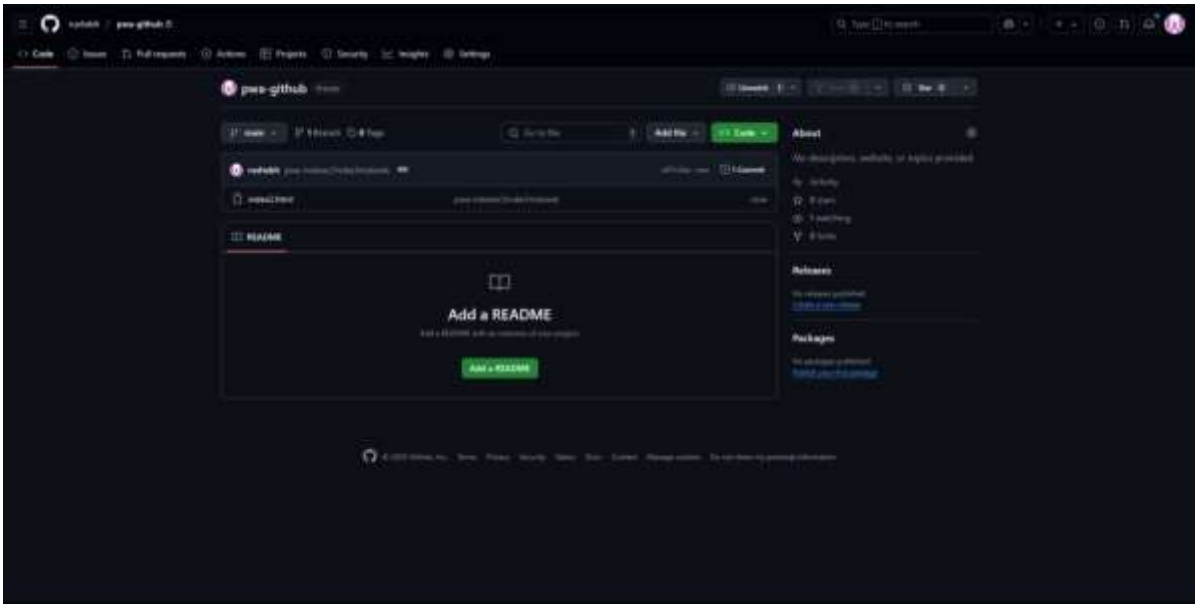


Step 2: Upload Your Website Files

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

You can either:

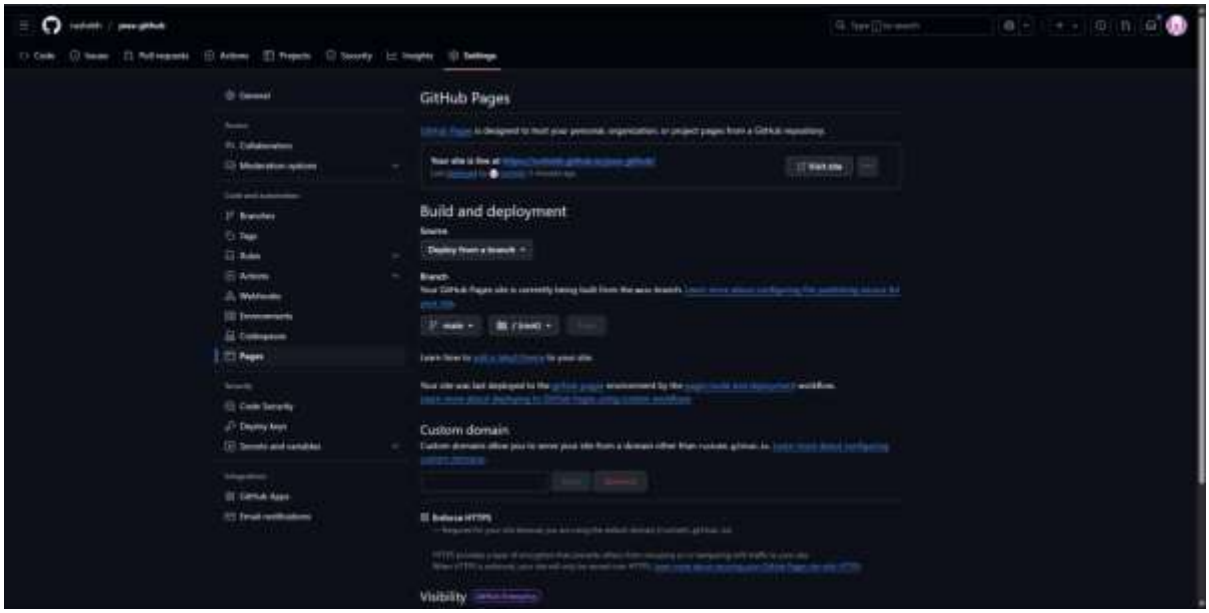
- Drag and drop files on the web interface



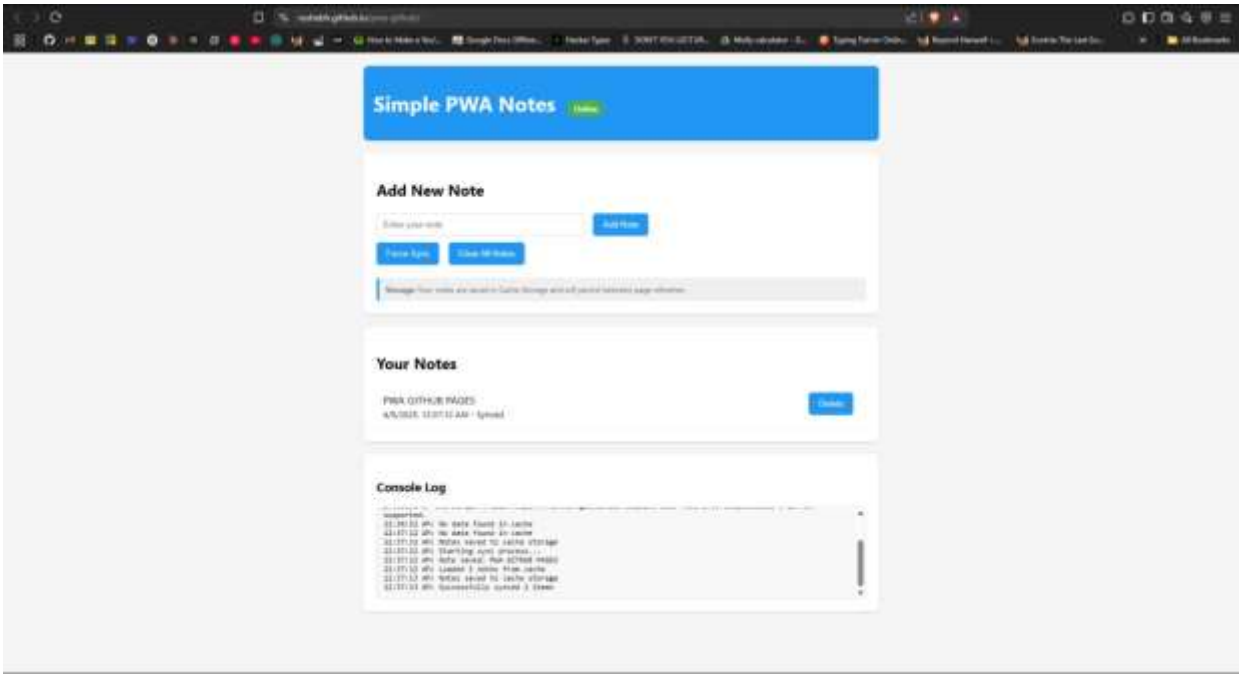
- Use Git commands (`git add`, `git commit`, `git push`) from your local machine

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



w

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.

Rushabh Jain
D15B - 23

A1

04

MPL/MAD - A-I

Q1 a)

→ Features

1) Single codebase for both Android & IOS
2) Hot Reload - Instantly see changes in the app w/o restring
3) Fast Performance - Uses dart language & a compiled approach for smooth & high performance
4) Open source - Backed by Google & a large developer community, ensuring continuous development

Advantages

1) Faster Development - Hot Reload & single codebase reduce development time
2) Cost-effective - Since same code runs on both Android & iOS, development cost is reduced
3) Reduce performance issues - App runs natively & is free of lags.

Q1 b)

→ 1) Single vs Separate Codebase -
- Trad Approach - Separate codebase for Android (Java/Kotlin) & iOS (Swift)
- Flutter - Single dart based codebase for both platforms.

2) Rendering Engine vs Native UI components
- Trad Approach - Relies on native UI components which can lead to inconsistencies
- Flutter - Uses Skia RA to draw everything from scratch.

Why Popular?

- Faster development with Hot Reload.
- Cross Platforms Efficiency
- Consistent UI Across Devices
- Improved Performance
- Easy Backend Integration

Q2

- Widget Tree in Flutter
  - It is a fundamental structure that represents the UI of the application. A hierarchial arrangement of widgets where each widget defines a component UI.
  - Flutter UI is directly entirely built using widgets which can be stateless or stateful. It determines how the UI should be rendered.

- Widget Composition –
  refers to building complex UIs by combining smaller, reusable widgets. i.e. flutter breaks down pages into small blocks that can be reused & nested with each other

Eg

```
class prof extends stateless widget {
    final string name;
    final string imgUrl;
    prof ({required this.name, required this.imgUrl});
    @ override
    Widget build (Build context length) {
        return Card (
            child: column(
```

children [
Image.network (img Url)
Sized box (h: 10)
Text (...)
]
}}

- Benefits of Widget Composition
  - Reusability - Can be reused at diff places
  - Maintability - easier to debug
  - Performance - rebuilds only necessary part of widget

1) Structural Widgets
   Eg - Material App, Scaffold, Container

Eg- Mtrl App (
home : Scaffold (
appBar : AppBar (title: Text ('Flutter Widget Tree')
body: Container (
padding: Edge
),
),
);

2) Interactive Widgets
   Eg - Textfield, Elevation Button, Gesture Detector

Elevate Button (
onPressed: () {
print ("Button Pressed")
};
})

3) Styling Widgets
eg. Text, Image, Icon, etc

```
column (
  children [
  Text ("Welcome")
  Image.network ("flutter/img1.jpg")
  ];
);
```

Q3 a)

⇒ - State refers to data that can change during its lifetime of an app -
Eg - Input, UI changes, etc.
- Types of State - 1) Ephemeral
2) Appwide State

- Importance
1) Code Maintainability & Scalability -
Managing state properly makes the code modular reusable & scalable

2) Efficient UI updates - UI is rebuilt when state changes, efficient state mgmt ensures only necessary parts are updated.

3) Data Consistency & Synchronisation
data remain consistent across different screens & widgets.

- setState - Local state
Pros - Simple & built in, easy to use
Cons - Not scalable

Scenario - When managing simple UI elements within a single widget like toggling dark mode

- Provider - App Wide State
Pros - Lightweight, efficient, flutter-recommended
Cons - Boiler plate code for nested providers

Scenario - When sharing state across multiple widgets like authentication

- Riverpod - App State
Pros - eliminates provider limitations, performance ↑
Cons - Requires learning new concepts

Scenario - when building a complex scalable app with global state management.

Q4
a)
→

Step 1 - Create a firebase project
• Firebase console → "Add project"

Step 2 - Register flutter app → Firebase
• FB dashboard → Add App → Select platform

Step 3 - Install FB dependencies
Add them in pubsec.yaml
firebase_core, auth
cloud_firestore
Run - flutter pub get

Step 4 Configure for Android & iOS
- classpath - com.google.gms:google-services:4.3.15
- Add 'com.google.gms.google-services' plugin

Step 5 Initialize Firebase in Flutter

- Benefits -
  - Easy to set Up & Scale
  - Provides multiple authentication
  - secure file cloud storage
  - Firebase Cloud Messaging

Q 4 b) Flutter Services -

⇒ 1) Firebase Authentication
Enables secure authentication using email / pass, Google, Fb, apple, etc.

2) Cloud Firestore
Stores & syncs data in real-time across devices, supports structured data, queries & offline access.

Eg -

```
Firebase.Firestore.instance.collection('users').add({
  ...
});
```

Rushabh Jain

- Realtime Databases
  A real-time JSON based DB that auto-updates the data across devices
  Ex -

```
DBReference.ref = ..... ("message")
ref.set(.....);
```

- Firebase Cloud Messaging
  Enables push notifications & messaging between users
  Ex -

```
FBMessaging.instance ..... ( );
```

- Analytics -
  Tracks User interactions & also app performance.
  Ex

```
FBAnalytics.analytics = .....
analytics.logEvent (..... *event* ..... );
```

- Firebase Hosting -
  Deploys & serves web applications securely with automatic SSL.

Data Synchronization in Firebase
FB ensures real-time data synchronization across multiple devices & platforms

1) Cloud Firestore Sync Mechanism
   Uses realtime listeners to update UI instantly when data changes

Ex- Firebase Firestore.instance.collection(users)....
for (var doc in snapshot.docs) {
:
}
});

2) Runtime Database Sync Mechanism
Uses persistent websocket connections for live updates.
Ex Database Reference ref = ...
ref.onValue.listen((event) {
:
});

3) Offline Data Sync
Firestore caches data locally & syncs changes when the device is online
Ex Firebase FS.instance.settings = Settings(...);

4) Used Functions for Auto updates.
Automates backend logic to trigger updates data changes

Rushabh Jain
D15B-23

MPL-2 (Assignment)

A PWA is a web application that combines both web & mobile apps to deliver a seamless experience

They are designed to be fast, reliable & engaging across various devices

-Significance -
- Platform Independence
- Improved Performance
- Offline functionality
- No app store dependencies
- Engaging User Experience

- Difference from traditional mobile apps (TMA)
- PWA is installed via browser while TMA are installed via playstore
- PWA takes up less storage
- PWA are fast & browser dependent >, TMA are optimized for hardwares
- PWA has limited offline access.

Responsiveness is an approach to ensure that web pages adapt to different screen sizes & orientations using flexible grids.

Importance -
- Ensures a consistent user experience across diffe-

-nt devices
. Eliminates need for multiple codebases for di
-esent devices
. Enhances usability by making content reach
able on screen

Responsive	Fluid	Adaptive
Uses CSS media qrs to adjst layout	Uses % for elemnts for scale	Uses predefined layou for diff screen siz
Highly flexible	Completely Flexible	Fixed at breakpoints
Efficient performance	Smooth Scaling	May cause layout shifts

Q3

→ Life cycle phases -

1) Registration
. A service worker is registered using JS in web ap
The browser checks if SW exists ; if new or updated, it moves to the next phase

```
if ('service worker' in navigator){
    navigator.serviceWorker.register('/sw.js')
then(() => console.log('registered')
}
```

2) Installation
Occurs when a service worker is first downloaded

- Executes install event & caches assets in this phase using cache.addAll()

```
self.addEventListener('install', event => {
    event.waitUntil(
        caches.open('v1').then(cache => {
    return cache.addAll
});
```

Activation

The 'activate' event installs after installation & ensures old caches cleared if necessary.
Takes control of page using self.clients.claim().

```
self.addEventListener('activate', event => {
    event.waitUntil(
    caches.keys().then(keys => {
    return Promise.all(keys.filter(key => key !== 'v1'))
});
}
```

3) Fetching & Updates

listens to fetch event to intercept network requests and updates the new service workes if found.

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

Q4

→ · Indexed DB is a low level, asynchronous database API that allows service workers to store & manage structured data efficiently.
· It is useful for handling large amounts of offline data, like user preferences, cached API responses, or application test.

- Uses of Indexed DB -
· Persistent storage - Data remains even after browser is closed
· Supports Complex Data - Can store objects, arrays & files
· Efficient Fetching - Faster access than Cache API
· Offline API caching - Store & retrieve API response for offline access
· Large Asset Storage - Caches images, videos & documents efficiently.