

EXP 1:

Aim: To install and configure flutter environment.

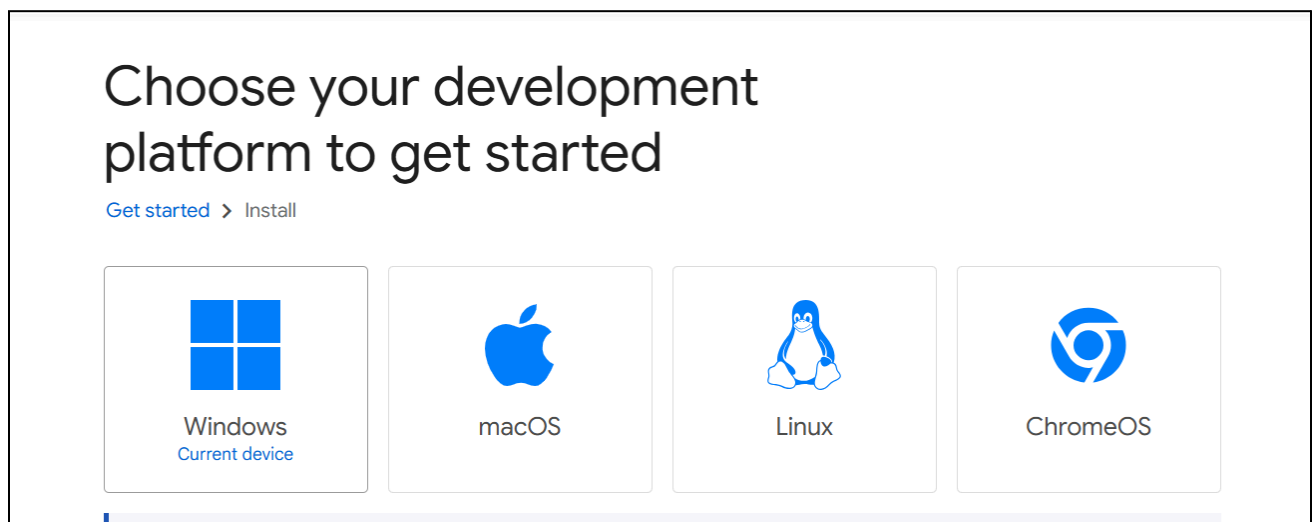
Theory:

Flutter: Flutter is an open source framework by Google for building beautiful, natively compiled, multi-platform applications from a single codebase.

- Fast: Flutter code compiles to ARM or Intel machine code as well as JavaScript, for fast performance on any device.
- Productive: Build and iterate quickly with Hot Reload. Update code and see changes almost instantly, without losing state.
- Flexible: Control every pixel to create customized, adaptive designs that look and feel great on any screen.

Steps performed:**Download the Flutter SDK**

Step 1: Go to the flutter official website and select your OS in order to get started.

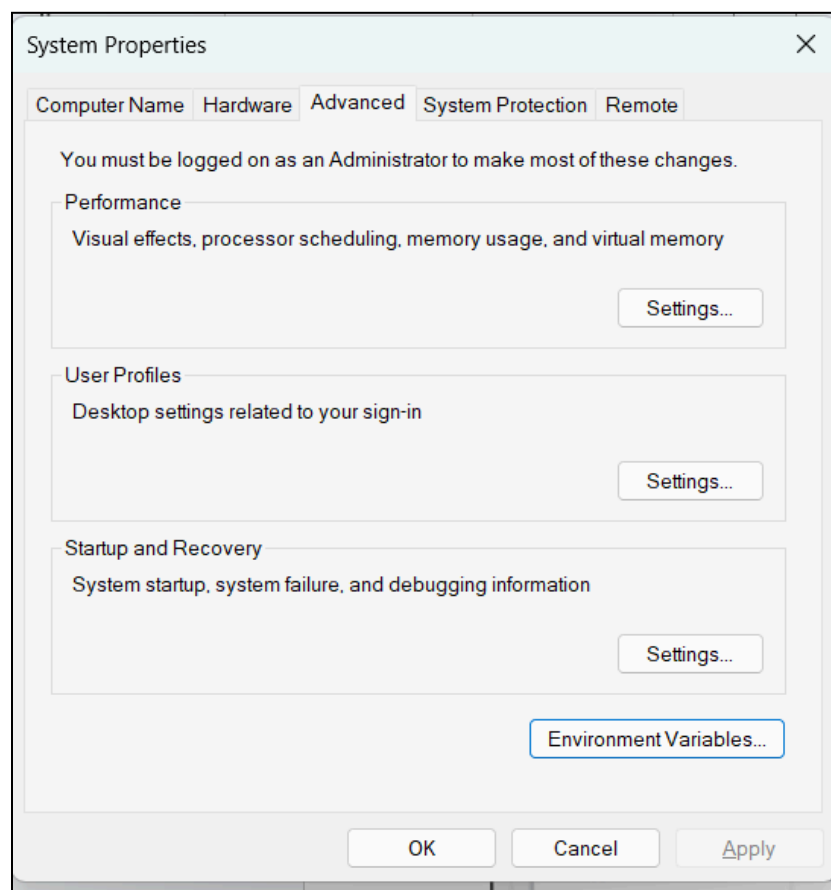


Step 2: Next, to download the latest Flutter SDK, click on the Windows icon. Here, you will find the download link for SDK.

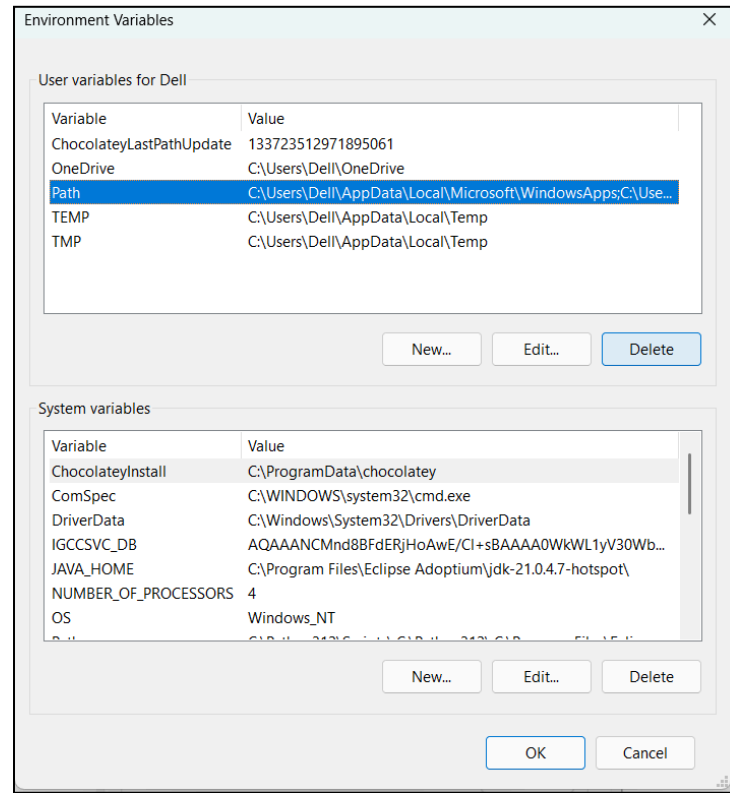
Step 3: When your download is complete, extract the zip file and place it in the desired installation folder or location.

Step 4: Now you will have to add flutter to the system path. To do so follow the below given steps:

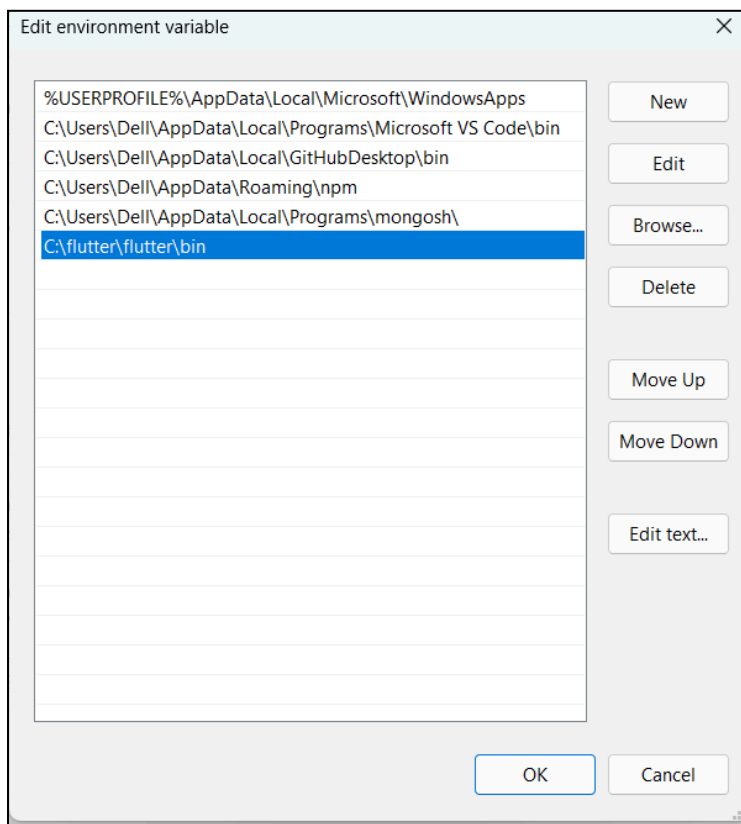
Step 4.1: Go to settings and search for environment variables, then select “Edit the system environment variables”. In the advanced tab, select the environment variables button.



Step 4.2: Now, in the next screen, select path and edit.



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



Step 5: Now, run the `flutter --version` command in command prompt to check if Flutter is correctly installed and see the version.

```
C:\Users\Dell>flutter --version
Flutter 3.29.2 • channel stable • https://github.com/flutter/flutter.git
Framework • revision c236373904 (3 weeks ago) • 2025-03-13 16:17:06 -0400
Engine • revision 18b71d647a
Tools • Dart 3.7.2 • DevTools 2.42.3
```

Now, run the `flutter doctor` command. This command checks for all the requirements of Flutter app development and displays a report of the status of your Flutter installation.

```
C:\Users\Dell>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 3.29.2, on Microsoft Windows [Version 10.0.26100.3476], locale en-IN)
[✓] Windows Version (11 Home Single Language 64-bit, 24H2, 2009)
[✓] Android toolchain - develop for Android devices (Android SDK version 35.0.1)
[✓] Chrome - develop for the web
[✓] Visual Studio - develop Windows apps (Visual Studio Community 2022 17.11.4)
[✓] Android Studio (version 2024.2)
[✓] VS Code (version 1.99.0)
[✓] Connected device (3 available)
[✓] Network resources

• No issues found!
```

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

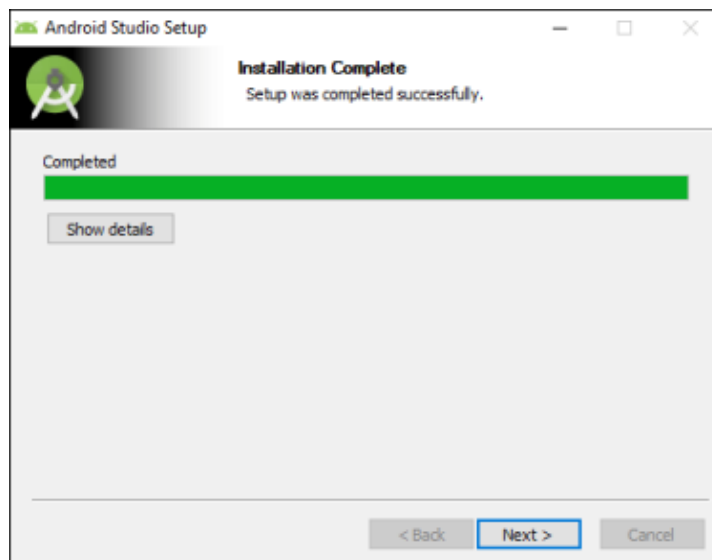
Step 7: Install the Android SDK. If the `flutter doctor` command does not find the Android SDK tool in your system, then you need to first install the Android Studio IDE. To install Android Studio IDE, follow the following steps.

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

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



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



Step 7.4: In the above screen, click Next-> Finish. Once the Finish button is clicked, you need to choose the 'Don't import Settings option' and click OK. It will start the Android Studio.

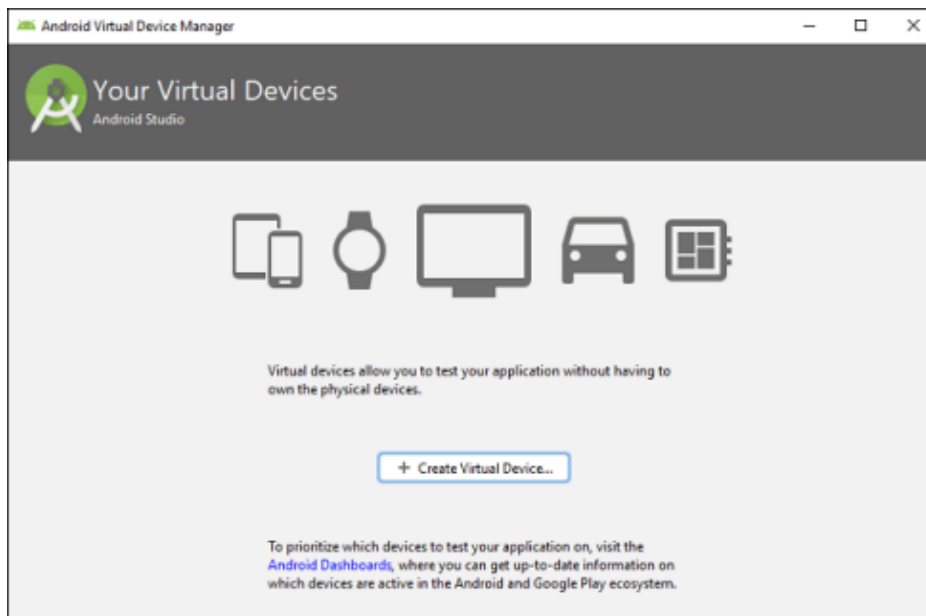
Step 7.5: Run the flutter doctor command again and run flutter doctor --android-licenses

command to accept all required Android SDK licenses needed for Flutter to build and run Android apps.

```
C:\Users\Dell>flutter doctor --android-licenses
Warning: Errors during XML parse:          ] 36% Fetch remote repository..
Warning: Additionally, the fallback loader failed to parse the XML.
[=====] 100% Computing updates...
All SDK package licenses accepted.
```

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

Step 8.1: To set an Android emulator, go to Android Studio > Tools > Android > AVD Manager and select Create Virtual Device. You will get the following screen.



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

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

Step 8.4: Now, verify the AVD configuration. If it is correct, click on Finish.

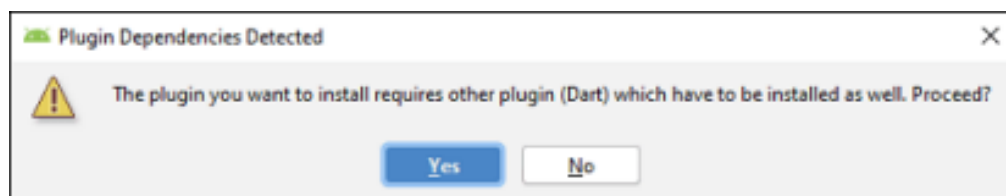
Step 8.5: Lastly, click on the icon pointed into the red color rectangle. The Android emulator will be displayed as shown in the below screen.



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

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

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



Step 9.3: Restart Android Studio.

Conclusion:

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

EXP 2

Aim: To design Flutter UI by including common widgets.

Theory:

Widgets:

Each element on a screen of the Flutter app is a widget. The view of the screen completely depends upon the choice and sequence of the widgets used to build the apps and the structure of the code of an app is a tree of widgets.

Category of Widgets:

There are mainly 14 categories in which the flutter widgets are divided. They are mainly segregated on the basis of the functionality they provide in a flutter application.

1. Accessibility: These are the set of widgets that make a flutter app more easily accessible.
2. Animation and Motion: These widgets add animation to other widgets.
3. Assets, Images, and Icons: These widgets take charge of assets such as display images and show icons.
4. Async: These provide async functionality in the flutter application.
5. Basics: These are the bundle of widgets that are absolutely necessary for the development of any flutter application.
6. Cupertino: These are the iOS designed widgets.
7. Input: This set of widgets provides input functionality in a flutter application.
8. Interaction Models: These widgets are here to manage touch events and route users to different views in the application.
9. Layout: This bundle of widgets helps in placing the other widgets on the screen as needed.
10. Material Components: This is a set of widgets that mainly follow material design by Google.
11. Painting and effects: This is the set of widgets that apply visual changes to their child widgets without changing their layout or shape.
12. Scrolling: This provides scrollability of to a set of other widgets that are not scrollable by default.
13. Styling: This deals with the theme, responsiveness, and sizing of the app.
14. Text: This displays text.

Description of few of the widgets are as follows:

- Scaffold– Implements the basic material design visual layout structure.
- App-Bar- To create a bar at the top of the screen.
- Text- To write anything on the screen.
- Container– To contain any widget.
- Center– To provide center alignment to other widgets.

Code:

Code for option buttons:

```
Widget _buildOptionButton(  
    BuildContext context,  
    String text,  
    IconData icon,  
    Color color, {  
    Color textColor = Colors.white,  
    Color? iconColor,  
    required Widget destination,  
}) {  
    return Container(  
        width: double.infinity,  
        height: 64,  
        decoration: BoxDecoration(  
            color: color,  
            borderRadius: BorderRadius.circular(28),  
            boxShadow: [  
                BoxShadow(  

```

```
        color: color.withOpacity(0.5),

        blurRadius: 10,

        offset: const Offset(0, 5),

    ),

],

border: Border.all(

    color: SelectionScreen.textDark.withOpacity(0.5),

    width: 1.5,

),

),

child: ElevatedButton(

    onPressed: () {

        Navigator.push(

            context,

            MaterialPageRoute(builder: (context) => destination),

        );

    },

style: ElevatedButton.styleFrom(

    backgroundColor: color,

    foregroundColor: textColor,

    elevation: 0,

    shape: RoundedRectangleBorder(

        borderRadius: BorderRadius.circular(28),

    ),

),
```

```
child: Row(  
  mainAxisAlignment: MainAxisAlignment.center,  
  children: [  
    Icon(  
      icon,  
      size: 26,  
      color: iconColor ?? textColor,  
    ),  
    const SizedBox(width: 12),  
    Text(  
      text,  
      style: TextStyle(  
        fontSize: 18,  
        fontWeight: FontWeight.w600,  
        letterSpacing: 0.5,  
        color: textColor,  
      ),  
    ),  
  ],  
,  
,  
,  
);  
}
```

Code for card widget:

```
Container(  
  padding: const EdgeInsets.symmetric(horizontal: 16, vertical: 14),  
  decoration: BoxDecoration(  
    color: deepMint.withOpacity(0.8),  
    borderRadius: BorderRadius.circular(16),  
    boxShadow: [  
      BoxShadow(  
        color: Colors.black.withOpacity(0.08),  
        blurRadius: 6,  
        offset: const Offset(0, 3),  
      ),  
    ],  
    border: Border.all(  
      color: SelectionScreen.textDark.withOpacity(0.5),  
      width: 1.5,  
    ),  
  ),  
  child: Column(  
    mainAxisAlignment: MainAxisAlignment.center,  
    crossAxisAlignment: CrossAxisAlignment.center,  
    children: [  
      Text(  
        "Did you know?",
```

```

        style: TextStyle(
          fontSize: 18,
          fontWeight: FontWeight.w600,
          color: SelectionScreen.textDark.withOpacity(0.8),
        ),
      ),
    const SizedBox(height: 8),
    Text(
      "Journaling daily can improve your wellbeing by 23%",
      style: TextStyle(
        fontSize: 14,
        color: SelectionScreen.textDark.withOpacity(0.85),
        fontWeight: FontWeight.w500,
      ),
    ),
  ],
),
)

```

Main.dart code:

```

import 'package:firebase_core/firebase_core.dart';
import 'package:flutter/material.dart';
import 'package:timezone/data/latest_all.dart' as tz;
import 'LoginScreen.dart'; // Import your HomeScreen

void main() async {
  WidgetsFlutterBinding.ensureInitialized();

  await Firebase.initializeApp(

```

```

options: FirebaseOptions(
  apiKey: "AIzaSyA9kxrbh6kBWblj2qeYr-gEgIQ-SBruAhQ",
  appId: "1:264673015498:web:03c5c23b9d471390fb6633",
  messagingSenderId: "264673015498",
  projectId: "moodlog-project",
),
);

tz.initializeTimeZones(); // ✅ Ensure timezone is initialized correctly
print("🔥 Firebase Initialized Successfully!");
print("🌍 Timezones Initialized!");

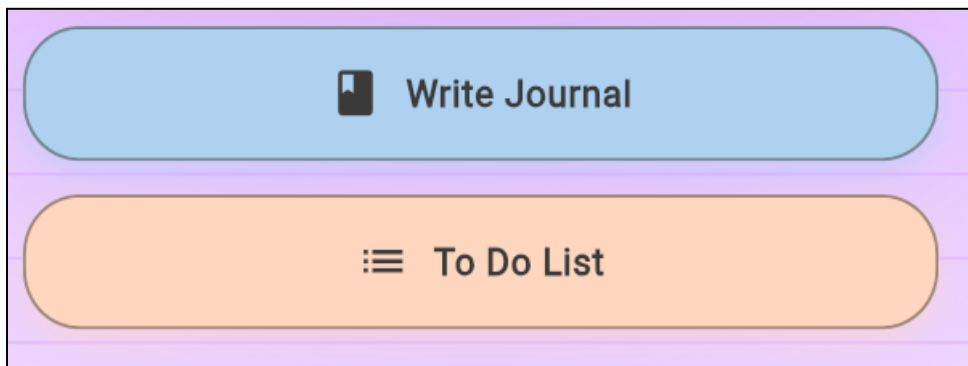
runApp(MyApp());
}

class MyApp extends StatelessWidget {
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      debugShowCheckedModeBanner: false,
      title: 'Mood Log',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: LoginScreen(), // Navigate to signup
    );
  }
}

```

Output:

Elevated buttons:



Save Mood

Weekly Summary

Text fields:

T

Title

Content

Checkbox:

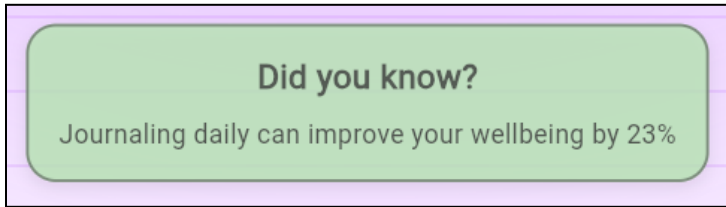
☐ complete project with report

☐ dmbi apriori experiment (17 april)

Container/card widgets:

hello

2025-04-07 21:14:18.535

**Conclusion:**

In conclusion, we can say that Flutter provides a rich set of widgets that are essential for creating intuitive and responsive UIs. Common widgets like `TextField`, `IconButton`, `ElevatedButton`, `Card`, and `Checkbox` play a key role in building the user interface of apps. These widgets help users input data, perform actions, display content in an organized manner, and enhance overall user interaction. Understanding and effectively using these widgets allows us to design clean, efficient, and user-friendly mobile applications.

EXP 3

Aim: To include icons, images, fonts in Flutter app

Theory:

In a Flutter application, visual elements like icons, images, and custom fonts enhance the user experience. These assets can be bundled within the app or loaded from external sources.

1. Icons are graphical representations of actions, objects, or ideas. In Flutter:
 - Material Icons are included by default and follow Material Design guidelines.
 - Icons are vector-based and scalable, making them resolution-independent.
 - Custom icon sets (like Font Awesome or custom SVG icons) can also be used by adding them as font files or via third-party packages.
 - Icons enhance UI clarity and visual appeal, especially in buttons, labels, or navigation elements.

2. Images in Flutter

Images are a key part of app UI and user experience. In Flutter:

- Asset Images are bundled with the app during development and included in the final build.
- Network Images are loaded from a URL and useful for dynamic or remote content.
- File Images are loaded from the local device storage, often used in apps that access user-generated photos.

3. Fonts in Flutter

Custom fonts let you personalize typography and branding in your app. In Flutter:

- You can include custom fonts by adding .ttf or .otf font files to the project.
- Fonts must be registered in the app's configuration file (usually pubspec.yaml).
- Custom fonts can be applied globally or to specific widgets for consistent styling.
- Flutter supports multiple font weights and styles (like bold, italic).

Code for homescreen.dart:

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

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

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

class _WelcomeScreenState extends State<WelcomeScreen> {
  static const Color pastelPink = Color(0xFFFF35E74);
  static const Color pastelPurple = Color(0xFFDAAAFF);
  static const Color pastelBlue = Color.fromARGB(255, 142, 205, 248);
  static const Color pastelGreen = Color.fromARGB(255, 56, 238, 56);
  static const Color textDark = Color(0xFF4A4A4A);

  Color selectedThemeColor = pastelPurple; // Default theme color

  @override
  Widget build(BuildContext context) {
    double screenWidth = MediaQuery.of(context).size.width;
    double screenHeight = MediaQuery.of(context).size.height;

    return Scaffold(
      body: Stack(
        children: [
          Positioned.fill(
            child: Container(
              decoration: BoxDecoration(
                gradient: LinearGradient(
                  colors: [selectedThemeColor.withOpacity(0.7), Colors.white],
                  begin: Alignment.topLeft,
                  end: Alignment.bottomRight,
                ),
              ),
            ),
          Positioned.fill(
            child: CustomPaint(
```

```

    painter: NotebookLinePainter(accentPink: selectedThemeColor),
  ),
),
SafeArea(
  child: Padding(
    padding: EdgeInsets.symmetric(horizontal: screenWidth * 0.05),
    child: Column(
      mainAxisAlignment: MainAxisAlignment.center,
      children: [
        SizedBox(height: screenHeight * 0.015),
        Column(
          mainAxisAlignment: MainAxisAlignment.center,
          crossAxisAlignment: CrossAxisAlignment.start, // Align text and icon to the left
          children: [
            Padding(
              padding: EdgeInsets.only(left: screenWidth * 0.05),
              child: Row(
                mainAxisAlignment: MainAxisAlignment.min,
                children: [
                  Text(
                    'MoodLog',
                    style: TextStyle(
                      fontFamily: 'DancingScript',
                      fontSize: 48,
                      color: textDark,
                      fontWeight: FontWeight.w700,
                    ),
                  ),
                  const SizedBox(width: 10),
                  Image.asset(
                    'icons/pencil.png',
                    width: 35,
                    height: 40,
                  ),
                ],
              ),
            ],
          ),
        ),
        const SizedBox(height: 16),
        SizedBox(
          height: 40,
          child: DefaultTextStyle(

```

```

style: TextStyle(
  fontFamily: 'DancingScript',
  fontSize: 25,
  color: textDark.withOpacity(0.9),
  fontWeight: FontWeight.w600,
  letterSpacing: 1.0,
  shadows: [
    Shadow(
      offset: Offset(0.5, 0.5), // Reduced offset for subtle effect
      blurRadius: 3.0, // Lower blur for softer shadow
      color: Colors.black.withOpacity(0.2), // Less opacity for cleaner look
    ),
  ],
),
child: AnimatedTextKit(
  key: ValueKey('welcomeText'),
  totalRepeatCount: 1,
  pause: Duration(milliseconds: 300),
  animatedTexts: [
    TypewriterAnimatedText(
      'Write your day, track your mood',
      speed: Duration(milliseconds: 100),
    ),
  ],
),
),
const SizedBox(height: 48),
Row(
  mainAxisAlignment: MainAxisAlignment.center,
  children: [
    Icon(Icons.star_border, color: pastelPink, size: 36),
    SizedBox(width: 24), // Kept spacing same
    Icon(Icons.favorite_border, color: pastelBlue, size: 36),
    SizedBox(width: 24),
    Icon(Icons.mood, color: pastelGreen, size: 36),
  ],
),
const SizedBox(height: 32),
Container(
  width: double.infinity,
  height: 56,
  decoration: BoxDecoration(

```

```

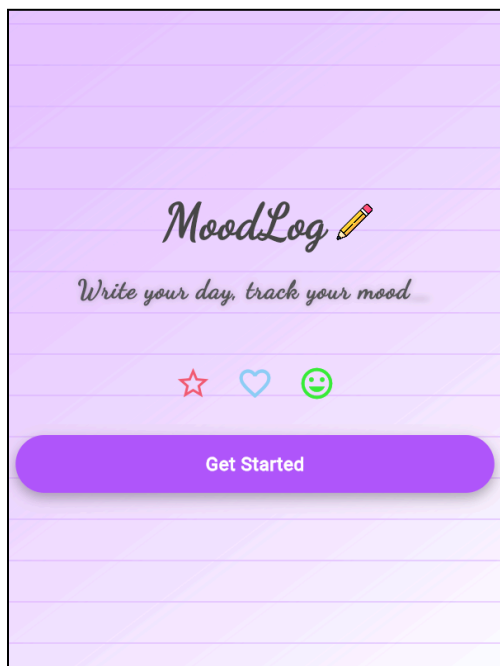
color: selectedThemeColor.withOpacity(0.85), // Slightly darkened
borderRadius: BorderRadius.circular(28),
boxShadow: [
  BoxShadow(
    color: Colors.black.withOpacity(0.3), // Improved shadow contrast
    blurRadius: 10,
    offset: Offset(0, 5),
  ),
],
),
child: ElevatedButton(
  onPressed: () {
    Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => SelectionScreen()),
    );
  },
  style: ElevatedButton.styleFrom(
    backgroundColor: const Color.fromARGB(255, 179, 85, 250),
    foregroundColor: Colors.white,
    elevation: 0,
    shape: RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(28),
    ),
  ),
  child: const Text(
    'Get Started',
    style: TextStyle(
      fontSize: 18,
      fontWeight: FontWeight.w600,
      letterSpacing: 0.5,
    ),
  ),
),
],
),
),
),
],
),
),
),
];
);
}

```

```
}
```

```
class NotebookLinePainter extends CustomPainter {  
    final Color accentPink;  
    NotebookLinePainter({required this.accentPink});  
  
    @override  
    void paint(Canvas canvas, Size size) {  
        final Paint paint = Paint()  
        ..color = accentPink.withOpacity(0.5)  
        ..strokeWidth = 1.5  
        ..style = PaintingStyle.stroke;  
  
        double lineSpacing = 40.0;  
        for (double i = 0; i < size.height; i += lineSpacing) {  
            canvas.drawLine(Offset(0, i), Offset(size.width, i), paint);  
        }  
    }  
  
    @override  
    bool shouldRepaint(covariant CustomPainter oldDelegate) {  
        return true;  
    }  
}
```

Output:



Conclusion:

In conclusion, the home page design of the app makes effective use of visual elements to create an engaging and journal-like experience. Icons such as flower, heart, and pencil enhance the visual appeal and intuitiveness of the interface. The background image of notebook lines reinforces the concept of journaling, while the use of the DancingScript font adds a handwritten, personal touch.

EXP 4

Aim: To create an interactive Form using form widget

Theory:

A Form Widget is a user interface element used to collect input from users. Forms are essential in web and application development, as they allow users to enter data such as login credentials, personal details, or feedback. Interactive forms enhance user experience by providing validation, responsiveness, and dynamic interactions.

Key Concepts in Interactive Forms

1. Form Widgets – These are components like text fields, checkboxes, radio buttons, dropdowns, and buttons that allow users to input data.
2. Validation – Ensuring that user input meets specific criteria (e.g., email format, required fields).
3. State Management – Forms can store data dynamically and update based on user interactions.
4. Event Handling – Capturing user actions like clicks, typing, or selections to trigger specific responses.
5. UI/UX Considerations – Forms should be visually appealing, easy to navigate, and provide clear feedback.

Implementation Approach

- Create a form using a programming language or framework (HTML, React, Flutter, etc.).
- Integrate form widgets such as input fields, dropdowns, and buttons.
- Implement validation to ensure correct user input.
- Provide real-time feedback through messages or UI changes.
- Ensure responsiveness across devices for a better user experience.

Code for New Journal Entry page:

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

import 'package:flutter/material.dart';

class NewEntryScreen extends StatefulWidget {

  @override

  _NewEntryScreenState createState() => _NewEntryScreenState();

}

class _NewEntryScreenState extends State<NewEntryScreen> {

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

  final TextEditingController _titleController = TextEditingController();

  final TextEditingController _contentController = TextEditingController();

  static const Color pastelPurple = Color(0xFFDAAAFF);

  static const Color pastelPink = Color(0xFFFF35E74);

  static const Color textDark = Color(0xFF4A4A4A);

  Future<void> saveNewEntry() async {

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

      try {

        FirebaseFirestore firestore = FirebaseFirestore.instance;
```

```
await firestore.collection('journalEntries').add({  
  'title': _titleController.text.trim(),  
  'content': _contentController.text.trim(),  
  'timestamp': FieldValue.serverTimestamp(),  
});
```

```
ScaffoldMessenger.of(context).showSnackBar(  
  SnackBar(  
    content: Text('New Entry Saved!'),  
    backgroundColor: Colors.pink.shade100,  
  ),  
);
```

```
_titleController.clear();  
_contentController.clear();  
Navigator.pop(context);  
} catch (e) {  
  print("Error saving new entry: $e");  
  ScaffoldMessenger.of(context).showSnackBar(  
    SnackBar(  
      content: Text('Failed to save the entry. Please try again.'),
```

```
        backgroundColor: pastelPink,

      ),

    );

  }

}

}
```

@override

```
Widget build(BuildContext context) {

  return Scaffold(

    appBar: AppBar(

      title: Text("Write New Journal Entry", style: TextStyle(color: textDark)),

      backgroundColor: Colors.purple.shade200,

      iconTheme: IconThemeData(color: textDark),

    ),

    body: SingleChildScrollView(

      child: Padding(

        padding: const EdgeInsets.all(16.0),

        child: Form(

          key: _formKey, // <--- New

          child: Column(

            crossAxisAlignment: CrossAxisAlignment.stretch,
```

```
children: [

  TextFormField(

    controller: _titleController,

    decoration: InputDecoration(

      labelText: 'Title',

      border: OutlineInputBorder(),

      prefixIcon: Icon(Icons.title, color: pastelPurple),

    ),

    validator: (value) {

      if (value == null || value.trim().isEmpty) {

        return 'Title cannot be empty';

      }

      return null;

    },

  ),

  SizedBox(height: 16),

  TextFormField(

    controller: _contentController,

    decoration: InputDecoration(

      labelText: 'Content',

      border: OutlineInputBorder(),

      alignLabelWithHint: true,
```

```
        prefixIcon: Icon(Icons.notes, color: pastelPurple),
      ),
      maxLines: 5,
      validator: (value) {
        if (value == null || value.trim().isEmpty) {
          return 'Content cannot be empty';
        }
        return null;
      },
    ),
    SizedBox(height: 20),
    ElevatedButton(
      onPressed: saveNewEntry,
      child: Text('Save Entry'),
    ),
  ],
),
),
),
),
),
);
}}
```

Output:

The image shows a mobile application interface for writing a new journal entry. At the top, there is a purple header bar with a back arrow icon on the left and the text 'Write New Journal Entry' in the center. Below the header, the form is divided into two main sections. The first section is a light purple box containing a text input field with a purple 'T' icon and the label 'Title'. The second section is a larger light purple box containing a text area with the label 'Content' and a purple icon of three horizontal lines. Below these sections is a rounded rectangular button with the text 'Save Entry'. The bottom half of the screen is a large, empty light purple area.

Conclusion:

In this experiment, a form was created using the Form widget to allow users to add new journal entries by entering a title and content. The use of TextFormField with validation made the form interactive and user-friendly. This helped ensure proper input before saving the data to Firestore, improving both functionality and user experience.

EXP 5

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

Theory:

Navigation and routing are essential for managing transitions between different screens in a Flutter app. Flutter uses a stack-based navigation model, where new routes (screens) are pushed onto the stack using the `Navigator.push()` method and removed using `Navigator.pop()`. Routes can be defined inline using `MaterialPageRoute` or globally using named routes for better structure in larger applications.

Routing ensures that the app is organized, scalable, and easy to navigate. It also helps manage screen transitions such as going from a home page to a details page or form screen.

Gestures in Flutter enable apps to respond to user input like tapping, swiping, dragging, or long pressing. Widgets like `GestureDetector`, `InkWell`, and `InkResponse` are used to detect and handle these gestures. For example, wrapping a `Container` in a `GestureDetector` allows it to respond to a tap or swipe, making the interface more interactive and user-friendly.

By combining navigation, routing, and gesture handling, a Flutter app becomes more engaging, intuitive, and functional for the user.

Code:**SelectionScreen.dart:**

```
import 'package:flutter/material.dart';
import 'package:mood_log/TrackMood.dart';
import 'package:mood_log/models/MoodEntry.dart';
import 'package:mood_log/todo_screen.dart';
import 'package:mood_log/JournalScreen.dart';

class SelectionScreen extends StatelessWidget {
  const SelectionScreen({Key? key}) : super(key: key);
```



```

@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Stack(
      children: [
        Positioned.fill(
          child: Container(
            decoration: BoxDecoration(
              gradient: RadialGradient(
                center: Alignment.topCenter,
                radius: 1.5,
                colors: [primaryColor.withOpacity(0.7), Colors.white],
                stops: const [0.2, 1.0],
              ),
            ),
          ),
        ),
        Positioned.fill(
          child: CustomPaint(
            painter: NotebookLinePainter(accentColor: primaryColor.withOpacity(0.4)),
          ),
        ),
        SafeArea(
          child: Padding(
            padding: const EdgeInsets.symmetric(horizontal: 24.0),
            child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              crossAxisAlignment: CrossAxisAlignment.center,
              children: [
                Text(
                  "What would you like to do?",
                  style: TextStyle(
                    fontFamily: 'DancingScript',
                    fontSize: 32,
                    color: textDark,
                    fontWeight: FontWeight.w700,
                    shadows: [
                      Shadow(
                        offset: const Offset(1.0, 1.0),
                        blurRadius: 3.0,
                        color: Colors.black.withOpacity(0.25),
                      ),
                    ],
                  ),
                ),
                const SizedBox(height: 30),
                _buildOptionButton(

```

```

context,
"Log your Mood",
Icons.edit,
softCoral,
textColor: textDark,
iconColor: const Color.fromARGB(255, 2, 2, 2),
destination: TrackMoodScreen(), // Add the target screen here
),
const SizedBox(height: 16),
_buildIconButton(
context,
"Write Journal",
Icons.book,
deepSkyBlue,
textColor: textDark,
destination: JournalScreen(), // Add the target screen here
),
const SizedBox(height: 16),
_buildIconButton(
context,
"To Do List",
Icons.list,
deepPeach,
textColor: textDark,
destination: ToDoScreen(), // Add the target screen here
),
const SizedBox(height: 30),
const SizedBox(height: 20),
Container(
padding: const EdgeInsets.symmetric(horizontal: 16, vertical: 14),
decoration: BoxDecoration(
color: deepMint.withOpacity(0.8),
borderRadius: BorderRadius.circular(16),
boxShadow: [
BoxShadow(
color: Colors.black.withOpacity(0.08),
blurRadius: 6,
offset: const Offset(0, 3),
),
],
border: Border.all(
color: textDark.withOpacity(0.5),
width: 1.5,
),
),
child: Column(
mainAxisAlignment: MainAxisAlignment.center,

```

```

crossAxisAlignment: CrossAxisAlignment.center,
children: [
  Text(
    "Did you know?",
    style: TextStyle(
      fontSize: 18,
      fontWeight: FontWeight.w600,
      color: textDark.withOpacity(0.8),
    ),
  ),
  const SizedBox(height: 8),
  Text(
    "Journaling daily can improve your wellbeing by 23%",
    style: TextStyle(
      fontSize: 14,
      color: textDark.withOpacity(0.85),
      fontWeight: FontWeight.w500,
    ),
  ),
],
),
),
],
),
),
],
),
),
],
),
);
}

```

```

Widget _buildOptionButton(
  BuildContext context,
  String text,
  IconData icon,
  Color color, {
  Color textColor = Colors.white,
  Color? iconColor,
  required Widget destination, // Destination screen as required parameter
}) {
  return Container(
    width: double.infinity,
    height: 64,
    decoration: BoxDecoration(
      color: color,
      borderRadius: BorderRadius.circular(28),
      boxShadow: [

```

```

BoxShadow(
  color: color.withOpacity(0.5),
  blurRadius: 10,
  offset: const Offset(0, 5),
),
],
border: Border.all(
  color: textDark.withOpacity(0.5),
  width: 1.5,
),
),
child: ElevatedButton(
  onPressed: () {
    // Navigate to the corresponding screen when the button is pressed
    Navigator.push(
      context,
      MaterialPageRoute(builder: (context) => destination),
    );
  },
  style: ElevatedButton.styleFrom(
    backgroundColor: color,
    foregroundColor: textColor,
    elevation: 0,
    shape: RoundedRectangleBorder(
      borderRadius: BorderRadius.circular(28),
    ),
  ),
  child: Row(
    mainAxisAlignment: MainAxisAlignment.center,
    children: [
      Icon(
        icon,
        size: 26,
        color: iconColor ?? textColor,
      ),
      const SizedBox(width: 12),
      Text(
        text,
        style: TextStyle(
          fontSize: 18,
          fontWeight: FontWeight.w600,
          letterSpacing: 0.5,
          color: textColor,
        ),
      ),
    ],
  ),
),

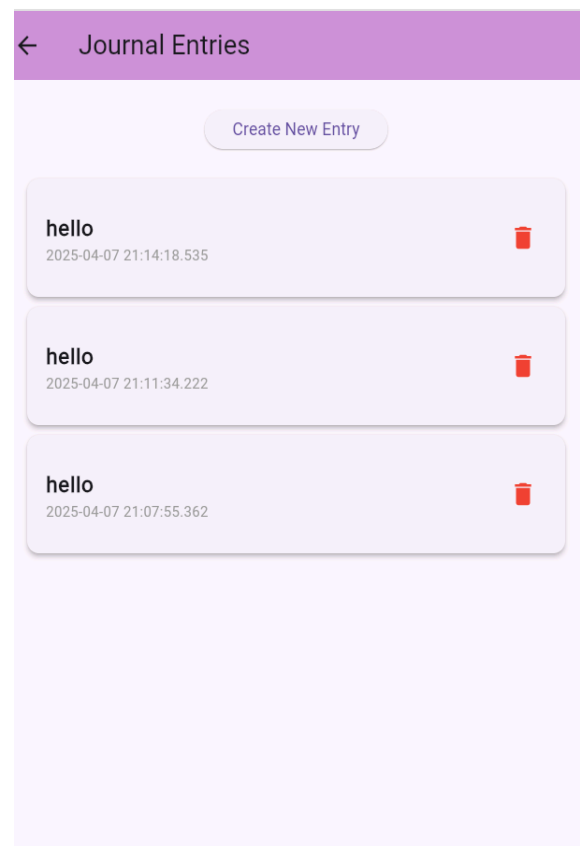
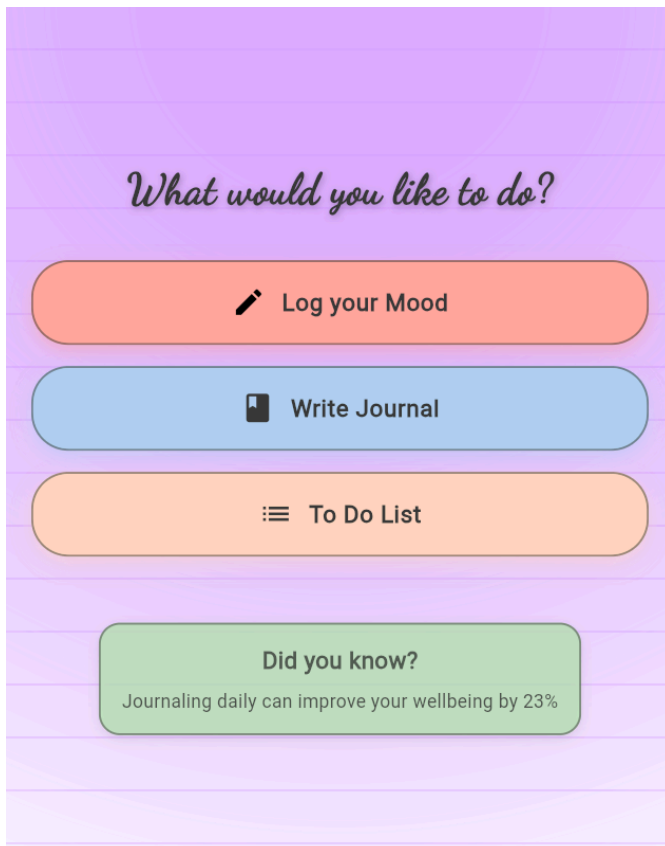
```

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

```
class NotebookLinePainter extends CustomPainter {  
    final Color accentColor;  
  
    const NotebookLinePainter({required this.accentColor});  
  
    @override  
    void paint(Canvas canvas, Size size) {  
        final Paint paint = Paint()  
        ..color = accentColor  
        ..strokeWidth = 1.2  
        ..style = PaintingStyle.stroke;  
  
        double lineSpacing = 40.0;  
        for (double i = 0; i < size.height; i += lineSpacing) {  
            canvas.drawLine(Offset(0, i), Offset(size.width, i), paint);  
        }  
    }  
  
    @override  
    bool shouldRepaint(covariant NotebookLinePainter oldDelegate) {  
        return oldDelegate.accentColor != accentColor;  
    }  
}
```

Output:

Clicking the write journal screen opens the Journal entries page:



Conclusion:

In conclusion, applying navigation, routing, and gestures in a Flutter app enhances user experience by enabling smooth transitions between screens, organized app structure through defined routes, and interactive features through gesture detection. These elements are essential for building intuitive and responsive mobile applications.

EXP 6

Aim: To connect flutter UI with firebase database

Theory:

Connecting Flutter UI with Firebase allows real-time cloud-based data management, enabling users to store, retrieve, and update data seamlessly from the app interface.

Key Concepts:

- **Firebase Integration:** The `firebase_core` package is used to initialize Firebase, while `cloud_firestore` connects the app to Firestore, Firebase's NoSQL cloud database.
- **Data Structure:** Firestore organizes data as collections (e.g., "journalEntries") and documents (each journal entry). This flexible, document-based structure suits dynamic apps like journals, chats, and to-do lists.
- **Data Operations:**
 - `add()` is used to insert new documents.
 - `get()` retrieves data.
 - `update()` modifies existing data.
 - `delete()` removes data from the collection.
- **Real-Time Updates:** Firestore supports real-time data syncing. Any changes made in the database can instantly reflect in the app using listeners like `StreamBuilder`.
- **Security & Rules:** Firebase provides authentication and Firestore security rules to control read/write access based on user roles or login status.
- **Scalability:** Being a cloud solution, Firebase scales automatically and is ideal for apps needing consistent performance and online syncing.

Code:**Signup screen:**

```
import 'package:flutter/material.dart';
import 'package:firebase_auth/firebase_auth.dart'; // Import Firebase Authentication package

class SignupScreen extends StatefulWidget {
```

```

const SignupScreen({super.key});

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

class _SignupScreenState extends State<SignupScreen> {
  // Controllers for TextFields
  final TextEditingController _nameController = TextEditingController();
  final TextEditingController _emailController = TextEditingController();
  final TextEditingController _passwordController = TextEditingController();

  // FirebaseAuth instance
  final FirebaseAuth _auth = FirebaseAuth.instance;

  // Sign Up method
  Future<void> _signUp() async {
    try {
      // Create user with email and password
      final UserCredential userCredential = await _auth.createUserWithEmailAndPassword(
        email: _emailController.text,
        password: _passwordController.text,
      );

      // Get the user object
      User? user = userCredential.user;

      // Send a verification email
      await user?.sendEmailVerification();

      // Notify the user
      ScaffoldMessenger.of(context).showSnackBar(
        const SnackBar(
          content: Text("Verification email sent! Please check your inbox."),
        ),
      );

      // Optionally navigate to the login screen
      Navigator.pop(context);
    } catch (e) {
      ScaffoldMessenger.of(context).showSnackBar(
        SnackBar(
          content: Text("Error: ${e.toString()}"),
        ),
      );
    }
  }
}

```



```

@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Padding(
      padding: const EdgeInsets.all(24.0),
      child: Center(
        child: Column(
          mainAxisAlignment: MainAxisAlignment.center,
          children: [
            Text(
              "Create an Account",
              style: TextStyle(
                fontSize: 28,
                fontWeight: FontWeight.bold,
                color: Colors.pinkAccent,
              ),
            ),
            const SizedBox(height: 30),

            // Full Name Field
            TextField(
              controller: _nameController,
              decoration: InputDecoration(
                labelText: "Full Name",
                border: OutlineInputBorder(),
              ),
            ),
            const SizedBox(height: 16),

            // Email Field
            TextField(
              controller: _emailController,
              decoration: InputDecoration(
                labelText: "Email",
                border: OutlineInputBorder(),
              ),
            ),
            const SizedBox(height: 16),

            // Password Field
            TextField(
              controller: _passwordController,
              obscureText: true,
              decoration: InputDecoration(
                labelText: "Password",
                border: OutlineInputBorder(),
              ),
            ),
          ],
        ),
      ),
    ),
  );
}

```

```

    ),
  ),
  const SizedBox(height: 24),

  // Sign Up Button
  ElevatedButton(
    onPressed: _signUp,
    style: ElevatedButton.styleFrom(
      backgroundColor: Colors.pinkAccent,
      foregroundColor: Colors.white,
      padding: const EdgeInsets.symmetric(vertical: 14, horizontal: 50),
      shape: RoundedRectangleBorder(
        borderRadius: BorderRadius.circular(20),
      ),
    ),
    child: Text("Sign Up", style: TextStyle(fontSize: 18)),
  ),










  const SizedBox(height: 20),

  // Navigate to Login Screen
  TextButton(
    onPressed: () {
      Navigator.pop(context); // Go back to Login screen
    },
    child: Text(
      "Already have an account? Login",
      style: TextStyle(color: Colors.pinkAccent),
    ),
  ),
],
),
),
),
);
}
}

```

Output:



 > journalEntries > ccxvrZoJcBrWH.			 More in Google Cloud 		
 (default)		 journalEntries  		 ccxvrZoJcBrWHHjGv3l 	
+ Start collection		+ Add document		+ Start collection	
journalEntries >		ccxvrZoJcBrWHHjGv3l >		+ Add field	
mood_entries		kZMf5uUx9MwXxHmnUUKn		content: "World"	
tasks		z1s65nuw7JV072Wv4eub		timestamp: 7 April 2025 at 21:07:55 UTC+5:30	
				title: "hello"	

Conclusion:

By connecting Flutter UI with Firebase Database, the app achieves real-time data management and secure user authentication. Using Firebase Authentication ensures safe login and registration, while Firestore enables efficient storage and retrieval of user data. This integration simplifies backend operations and enhances the responsiveness and interactivity of the Flutter application.

EXP 7

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

Theory:-

- Regular Web App

A regular web app is a website that is designed to be accessible on all mobile devices such that the content gets fit as per the device screen. It is designed using a web technology stack (HTML, CSS, JavaScript, Ruby, etc.) and operates via a browser. They offer various native-device features and functionalities. However, it entirely depends on the browser the user is using. In other words, it might be possible that you can access a native-device feature on Chrome but not on Safari or Mozilla Firefox because the browsers are incompatible with that feature.

- Progressive Web App

Progressive Web App (PWA) is a regular web app, but some extras enable it to deliver an excellent user experience. It is a perfect blend of desktop and mobile application experience to give both platforms to the end-users.

- Difference between PWAs vs. Regular Web Apps:

A Progressive Web is different and better than a Regular Web app with features like:

1. Native Experience

Though a PWA runs on web technologies (HTML, CSS, JavaScript) like a Regular web app, it gives user experience like a native mobile application. It can use most native device features, including push notifications, without relying on the browser or any other entity. It offers a seamless and integrated user experience that it is quite tough for one to differentiate between a PWA and a Native application by considering its look and feel.

2. Ease of Access

Unlike other mobile apps, PWAs do not demand longer download time and make memory space available for installing the applications. The PWAs can be shared and installed by a link, which cuts down the number of steps to install and use. These applications can easily keep an app icon on the

user's home screen, making the app easily accessible to the users and helps the brands remain in the users' minds, and improving the chances of interaction.

3. Faster Services

PWAs can cache the data and serve the user with text stylesheets, images, and other web content even before the page loads completely. This lowers the waiting time for the end-users and helps the brands improve the user engagement and retention rate, which eventually adds value to their business.

4. Engaging Approach

As already shared, the PWAs can employ push notifications and other native device features more efficiently. Their interaction does not depend on the browser user uses. This eventually improves the chances of notifying the user regarding your services, offers, and other options related to your brand and keeping them hooked to your brand. In simpler words, PWAs let you maintain the user engagement and retention rate.

5. Updated Real-Time Data Access

Another plus point of PWAs is that these apps get updated on their own. They do not demand the end-users to go to the App Store or other such platforms to download the update and wait until installed.

In this app type, the web app developers can push the live update from the server, which reaches the apps residing on the user's devices automatically. Therefore, it is easier for the mobile app developer to provide the best of the updated functionalities and services to the end-users without forcing them to update their app.

6. Discoverable

PWAs reside in web browsers. This implies higher chances of optimizing them as per the Search Engine Optimization (SEO) criteria and improving the Google rankings like that in websites and other web apps.

7. Lower Development Cost

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

The main features are:

1. Progressive - They work for every user, regardless of the browser chosen because they are built at the base with progressive improvement principles.
2. Responsive - They adapt to the various screen sizes: desktop, mobile, tablet, or dimensions that can later become available.
3. App-like - They behave with the user as if they were native apps, in terms of interaction and navigation.
4. Updated - Information is always up-to-date thanks to the data update process offered by service workers.
5. Secure - Exposed over HTTPS protocol to prevent the connection from displaying information or altering the contents.
6. Searchable - They are identified as “applications” and are indexed by search engines.
7. Reactivable - Make it easy to reactivate the application thanks to capabilities such as web notifications.
8. Installable - They allow the user to “save” the apps that he considers most useful with the corresponding icon on the screen of his mobile terminal (home screen) without having to face all the steps and problems related to the use of the app store.
9. Linkable - Easily shared via URL without complex installations.
10. Offline - Once more it is about putting the user before everything, avoiding the usual error message in case of weak or no connection. The PWA are based on two particularities: first of all the ‘skeleton’ of the app, which recalls the page structure, even if its contents do not respond and its elements include the header, the page layout, as well as an illustration that signals that the page is loading.

Code:

Manifest.json:

```
{  
  "name": "moodlog_project",  
  "short_name": "moodlog",  
  "start_url": "/",  
  "display": "standalone",  
  "background_color": "#0175C2",  
  "theme_color": "#0175C2",  
  "description": "App for journaling and mood tracking",  
  "orientation": "portrait-primary",  
  "prefer_related_applications": false,  
  "icons": [  

```

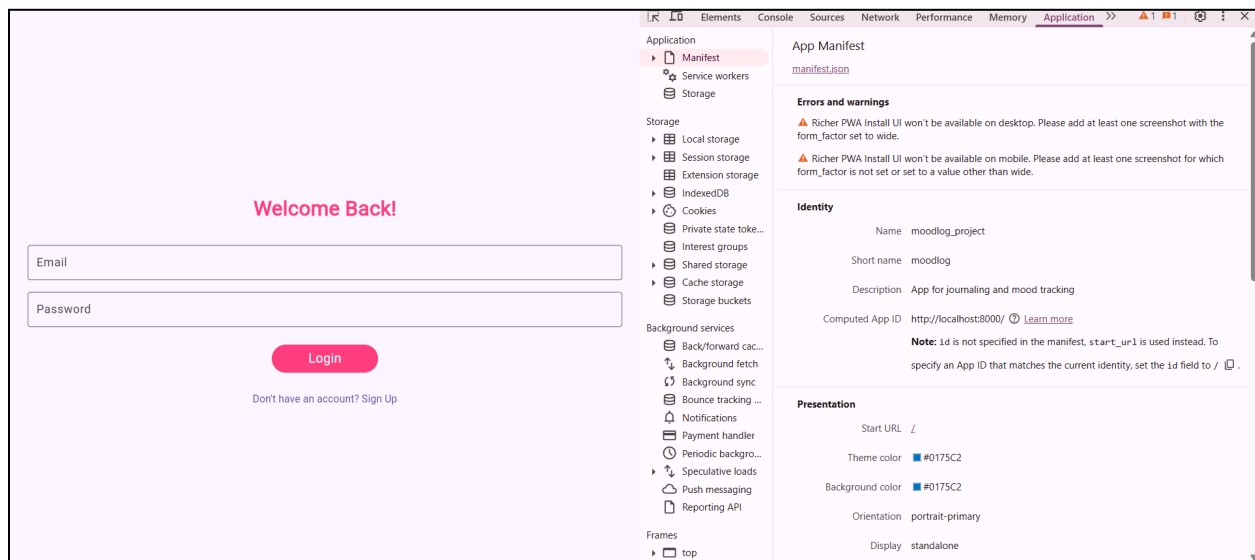
```
{
  "src": "icons/Icon-192.png",
  "sizes": "192x192",
  "type": "image/png",
  "purpose": "any"
},
{
  "src": "icons/Icon-512.png",
  "sizes": "512x512",
  "type": "image/png",
  "purpose": "any"
},
{
  "src": "icons/Icon-maskable-192.png",
  "sizes": "192x192",
  "type": "image/png",
  "purpose":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

Add the link tag to link to the manifest.json file

```
<title>Moodlog</title>
```

```
<link rel="manifest" href="manifest.json">
```

Output:



Conclusion:

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

EXP 8

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

Theory:

A Service Worker is a background script that runs separately from the web page and enables key Progressive Web App (PWA) features such as offline access, background sync, and push notifications.

Coding and Registering a Service Worker:

- The service worker is written in a separate JavaScript file (usually service-worker.js).
- It is registered in the main HTML or JavaScript file using `navigator.serviceWorker.register()`.
- This registration allows the browser to recognize and manage the service worker.

Installation and Activation:

- **Install Phase:** This is the first step when the browser detects a new service worker. In this phase, static assets (HTML, CSS, JS, images) are cached using `caches.open()` and `cache.addAll()` for offline availability.
- **Activate Phase:** After installation, the service worker moves to the activate phase. Here, old caches can be cleaned up using `caches.keys()` and `caches.delete()` to ensure only the latest resources are stored.

Role in an E-Commerce PWA:

- Ensures offline functionality so users can browse cached product pages without the internet.
- Improves performance through cached responses.
- Provides a reliable user experience, especially in low or no network areas.

Code:

flutter_service_worker.js

```
self.addEventListener('install', (event) => {  
  
    console.log('[Service Worker] Install event');  
  
    self.skipWaiting(); // Activate worker immediately  
  
});  
  
self.addEventListener('activate', (event) => {  
  
    console.log('[Service Worker] Activate event');  
  
    // Cleanup old caches here if needed  
  
});  
  
self.addEventListener('fetch', (event) => {  
  
    event.respondWith(  
  
        caches.match(event.request).then((response) => {  
  
            return response || fetch(event.request);  
  
        })  
  
    );  
  
});
```

index.html

```
<!DOCTYPE html>
```

```
<html>
```

```
<head>
```

```
  <base href="$FLUTTER_BASE_HREF">
```

```
  <meta charset="UTF-8">
```

```
  <meta content="IE=Edge" http-equiv="X-UA-Compatible">
```

```
  <meta name="description" content="A new Flutter project.">
```

```
  <!-- iOS meta tags & icons -->
```

```
  <meta name="apple-mobile-web-app-capable" content="yes">
```

```
  <meta name="apple-mobile-web-app-status-bar-style" content="black">
```

```
  <meta name="apple-mobile-web-app-title" content="moodlog">
```

```
  <link rel="apple-touch-icon" href="icons/Icon-192.png">
```

```
  <meta name="google-signin-client_id"
content="15128894708-pqnk893c2cns1s1du7g1ghk33dech5vm.apps.googleusercontent.com">
```

```
  <script src="https://accounts.google.com/gsi/client" async defer></script>
```

```
  <!-- Favicon -->
```

```
  <link rel="icon" type="image/png" href="favicon.png"/>
```

```
  <title>Moodlog</title>
```

```
  <link rel="manifest" href="manifest.json">
```

```
</head>
```

```
<body>
```

```
  <script src="flutter_bootstrap.js" async></script>
```

```

</body>

<script>

  if ('serviceWorker' in navigator) {

    window.addEventListener('load', function () {

      navigator.serviceWorker.register('flutter_service_worker.js')

        .then(function (registration) {

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

        }).catch(function (error) {

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

        });

    });

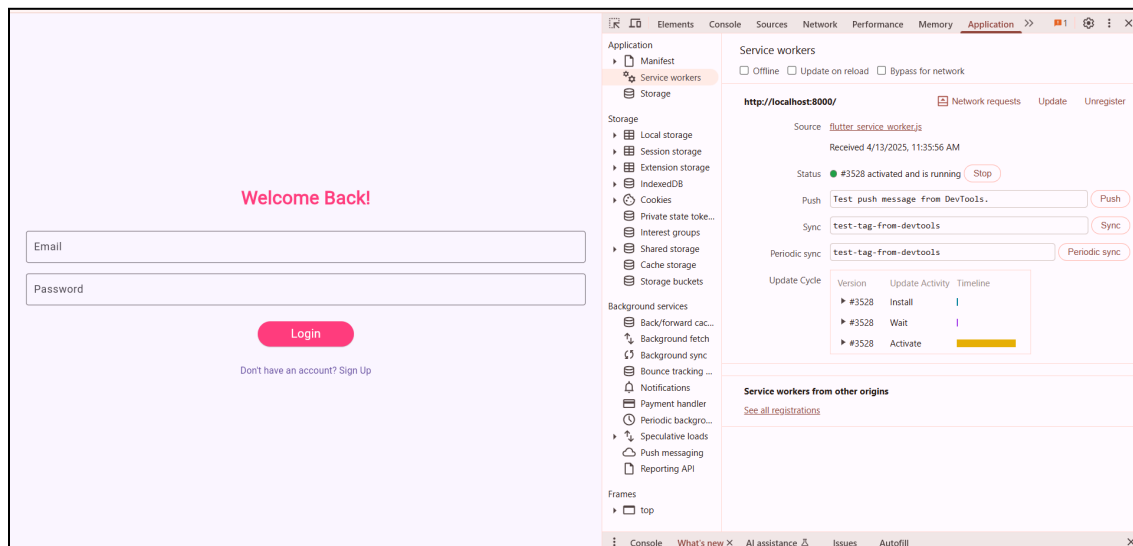
  }

</script>

</html>

```

Output:



Conclusion:

In this experiment, a service worker was successfully coded, registered, installed, and activated for the PWA. This enhanced the app's performance by enabling offline access, faster loading, and improved reliability, ensuring a seamless user experience even with limited or no internet connectivity.

EXP 9

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

Theory:**Service Worker**

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

Things to note about Service Worker:

1. A service worker is a programmable network proxy that lets you control how network requests from your page are handled.
2. Service workers only run over HTTPS. Because service workers can intercept network requests and modify responses, "man-in-the-middle" attacks could be very bad.
3. The service worker becomes idle when not in use and restarts when it's next needed. You cannot rely on a global state persisting between events. If there is information that you need to persist and reuse across restarts, you can use IndexedDB databases.
4. Service workers make extensive use of promises, so if you're new to promises, then you should stop reading this and check out Promises, an introduction.

- **Fetch Event**

You can track and manage page network traffic with this event. You can check existing cache, manage “cache first” and “network first” requests and return a response that you want.

Of course, you can use many different methods but you can find in the following example a “cache first” and “network first” approach. In this example, if the request’s and current location’s origin are the same (Static content is requested.), this is called “cacheFirst” but if you request a targeted external URL, this is called “networkFirst”.

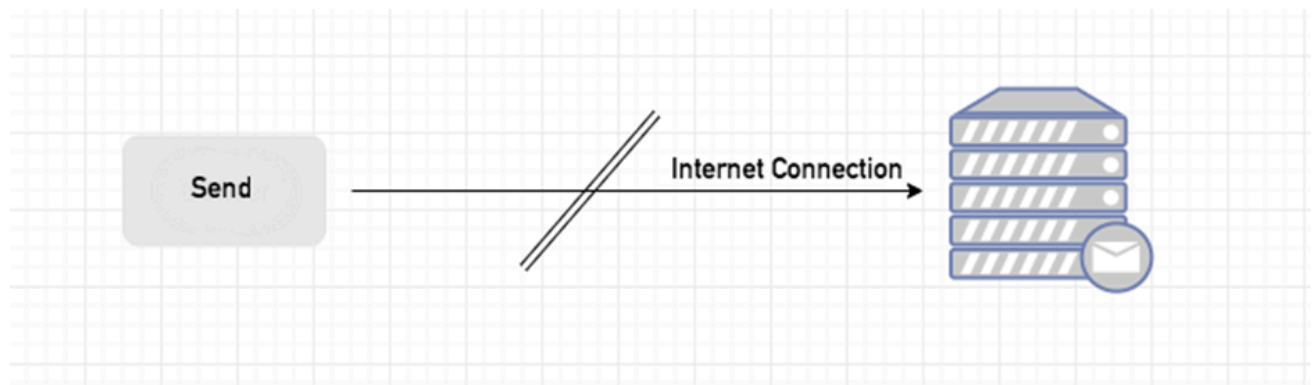
1. CacheFirst - In this function, if the received request has cached before, the cached response is returned to the page. But if not, a new response requested from the network.
2. NetworkFirst - In this function, firstly we can try getting an updated response from the network, if this process completed successfully, the new response will be cached and returned. But if this process fails, we check whether the request has been cached before or not. If a cache exists, it is returned to the page, but if not, this is up to you. You can return dummy content or information messages to the page.

- Sync Event

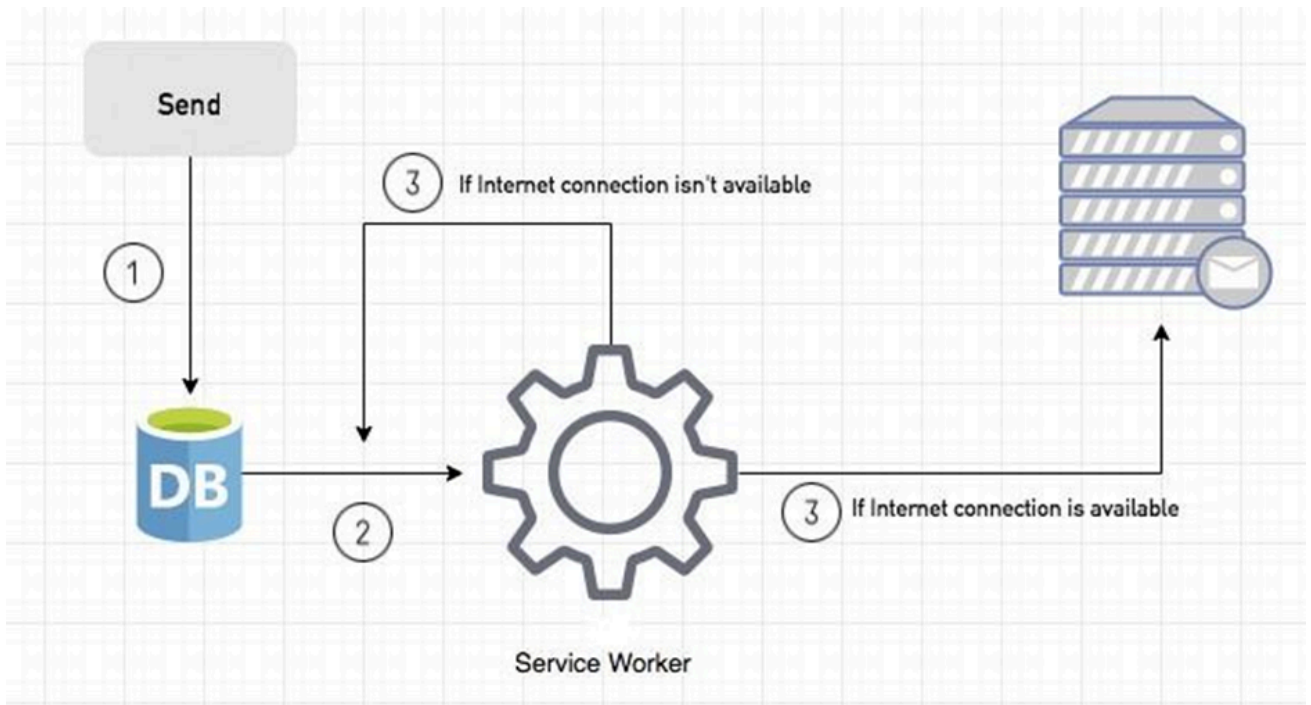
Background Sync is a Web API that is used to delay a process until the Internet connection is stable. We can adapt this definition to the real world; there is an e-mail client application that works on the browser and we want to send an email with this tool. Internet connection is broken while we are writing e-mail content and we didn't realize it. When completing the writing, we click the send button.

Here is a job for the Background Sync.

The following view shows the classical process of sending email to us. If the Internet Connection is broken, we can't send any content to Mail Server.



Here, you can create any scenario for yourself. A sample is in the following for this case.



1. When we click the “send” button, email content will be saved to IndexedDB.
2. Background Sync registration.
3. If the Internet connection is available, all email content will be read and sent to Mail Server.

If the Internet connection is unavailable, the service worker waits until the connection is available even though the window is closed. When it is available, email content will be sent to Mail Server.

You can see the working process within the following code block.

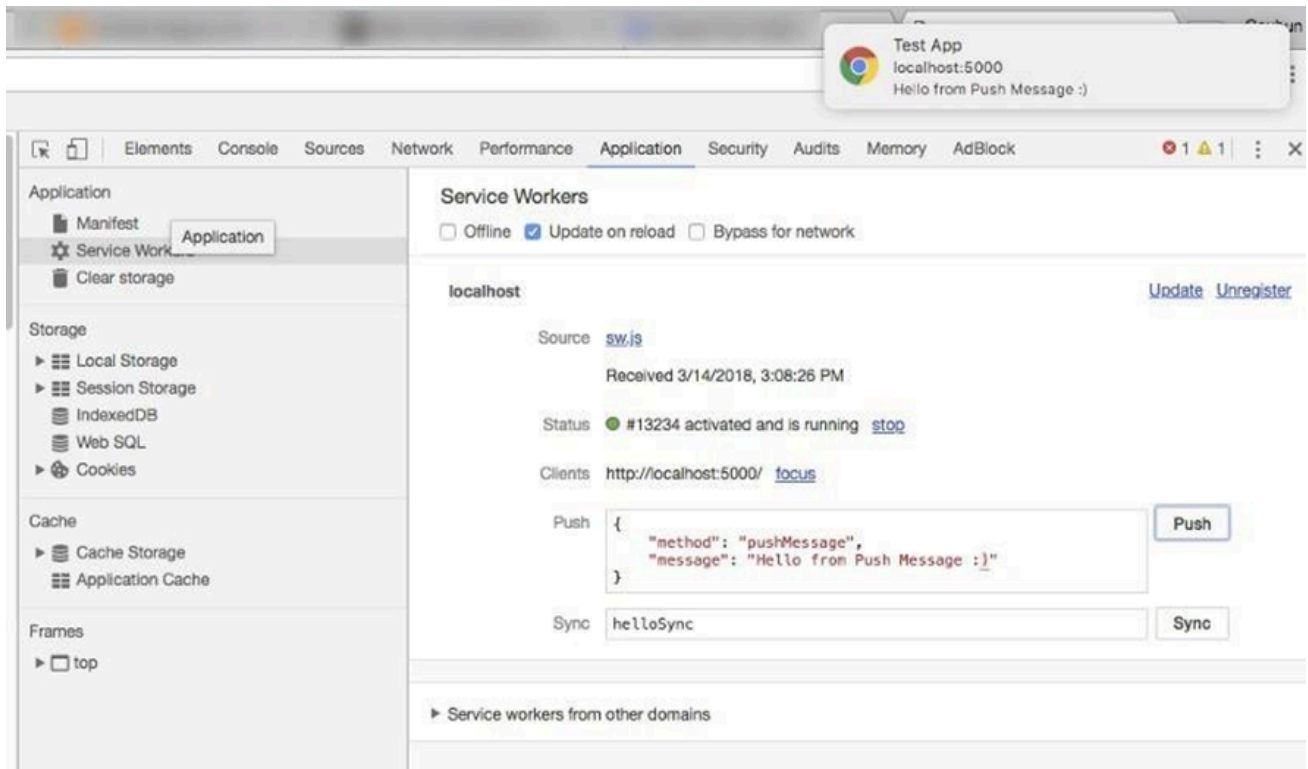
- Push Event

This is the event that handles push notifications that are received from the server. You can apply any method with received data.

We can check in the following example.

“Notification.requestPermission();” is the necessary line to show notification to the user. If you don’t want to show any notification, you don’t need this line.

You can use Application Tab from Chrome Developer Tools for testing push notification.



Code:

```
// Install event
self.addEventListener('install', (event) => {
  event.waitUntil(
    caches.open('my-cache').then((cache) => {
      return cache.addAll([
        '/', // Add your important files like index.html
        '/styles.css', // Add CSS files
        '/script.js', // Add JS files
        '/images/logo.png', // Add image files
      ]);
    })
  );
});
```

```
});
```

```
// Fetch event - Handle requests and serve cached resources
```

```
self.addEventListener('fetch', (event) => {  
  event.respondWith(  
    caches.match(event.request).then((cachedResponse) => {  
      if (cachedResponse) {  
        return cachedResponse; // Serve from cache  
      }  
      return fetch(event.request); // Fetch from network  
    })  
  );  
});
```

```
// Sync event - Handle background sync tasks
```

```
self.addEventListener('sync', (event) => {  
  if (event.tag === 'syncData') {  
    event.waitUntil(syncData()); // Function to sync data in the background  
  }  
});
```

```
function syncData() {  
  return fetch('/sync-endpoint', {  
    method: 'POST',  
    body: JSON.stringify({ data: 'syncData' }),  
    headers: { 'Content-Type': 'application/json' },  
  })  
  .then(response => response.json())
```

```

.then(data => console.log('Data synced:', data))

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

// Push event - Handle push notifications
self.addEventListener('push', (event) => {

  const options = {

    body: event.data.text(), // Notification body

    icon: '/images/notification-icon.png', // Icon for the notification

    badge: '/images/badge.png', // Badge for the notification

  };

  event.waitUntil(

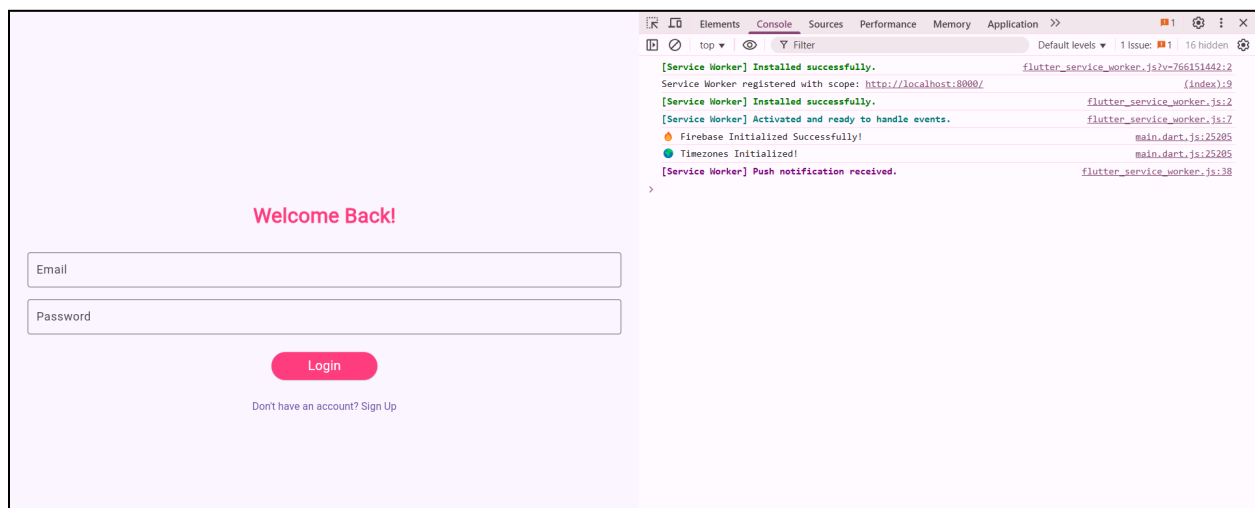
    self.registration.showNotification('New Offer!', options) // Show the notification

  );

});

```

Output:



Conclusion:

In conclusion, implementing service worker events like fetch, sync, and push significantly enhances the functionality of a PWA. The fetch event improves load speed and offline capabilities by caching assets, the sync event ensures background data synchronization when the user is back online, and the push event enables timely notifications to keep users engaged. These features provide a smoother, more reliable user experience, even in low or no network conditions, making the app more efficient and user-friendly.

EXP 10

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

Theory:

Deploying an E-commerce PWA built using Flutter to GitHub Pages involves converting the Flutter project into static web assets and hosting them on GitHub. Flutter compiles the UI and logic into HTML, CSS, and JavaScript files stored in the build/web directory. GitHub Pages is a free hosting platform provided by GitHub for serving static content directly from a repository.

To deploy:

1. A new Git branch (commonly gh-pages) is created to hold the web build.
2. The contents of build/web are added to this branch.
3. The repository is then configured to serve this branch via GitHub Pages.

This process makes the PWA publicly accessible through a GitHub URL and allows users to interact with it without any backend server.

Features:

1. Hosts static content from the GitHub repository.
2. Automatically deploys upon each push to the repository.
3. Supports Jekyll for static site generation.
4. Offers custom domain configuration with simple DNS setup.

Advantages:

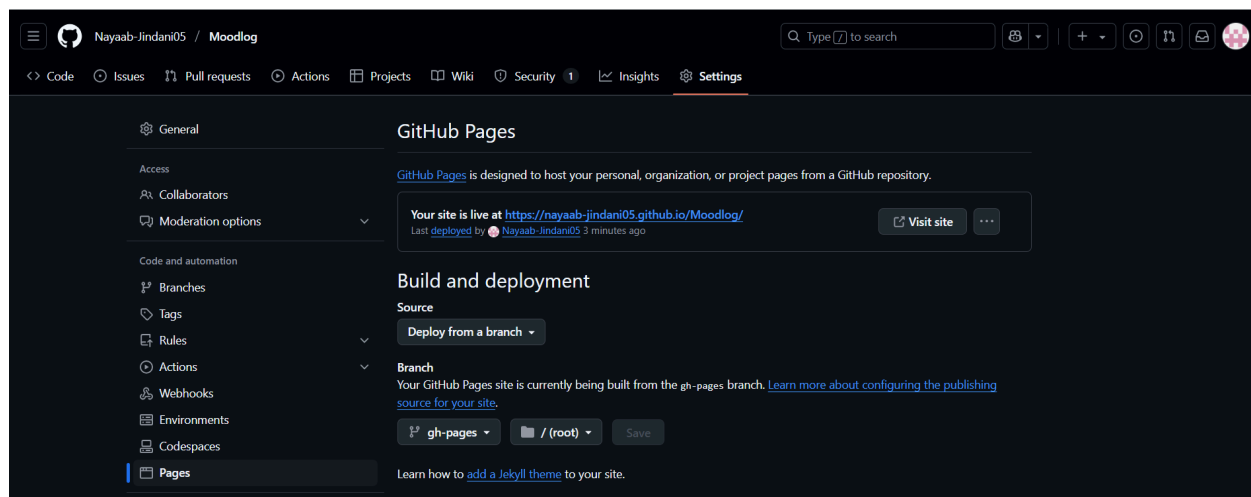
1. Easy to set up for users familiar with GitHub.
2. Native support for static site generators like Jekyll.
3. Custom domain support using a CNAME file.
4. No command-line tools required for basic deployments.

Disadvantages:

1. The website's code must be public unless a paid private repository is used.
2. Limited plugin support when using Jekyll.
3. Historically lacked HTTPS support for custom domains (now available when using GitHub's DNS).

Output:

```
PS C:\flutterapps\moodlog_project\build\web> cd build/web
>> git init
>> git remote add origin https://github.com/nayaab-jindani05/Moodlog.git
>> git checkout -b gh-pages
>> git add .
>> git commit -m "Deploy Flutter Web to GitHub Pages"
>> git push origin gh-pages --force
```



Conclusion:

In this experiment, the deployment of the Flutter-based PWA was successfully achieved using GitHub Pages. By initializing a Git repository inside the build/web directory and pushing the content to the gh-pages branch of the GitHub repository, the app became publicly accessible online. This method provides a simple and free hosting solution for Flutter web applications.

EXP 11

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

Theory:

Google Lighthouse is an open-source automated tool used to analyze the performance and quality of web apps, including Progressive Web Apps (PWA). It checks how well your app meets PWA standards such as offline support, service worker functionality, responsive design, fast load times, and installability.

By running Lighthouse in Chrome DevTools, developers get a detailed report highlighting strengths and improvement areas related to PWA features like:

- Service worker registration
- Manifest file validity
- HTTPS usage
- Performance and accessibility
- App install prompt availability

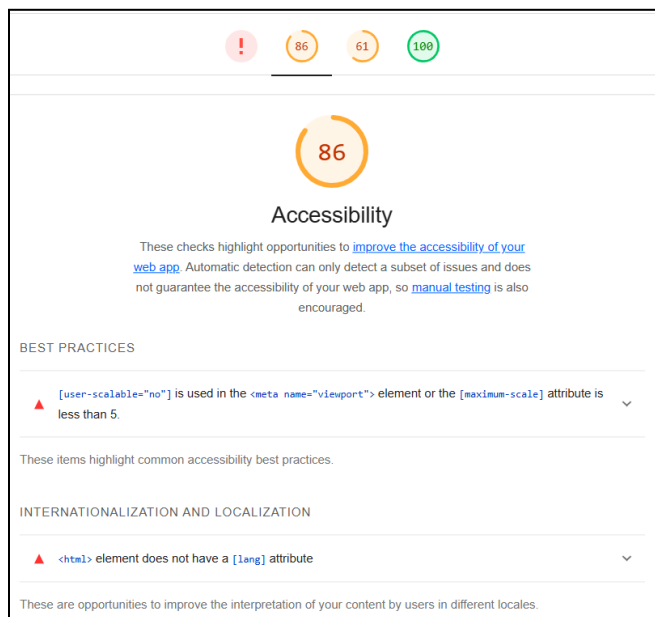
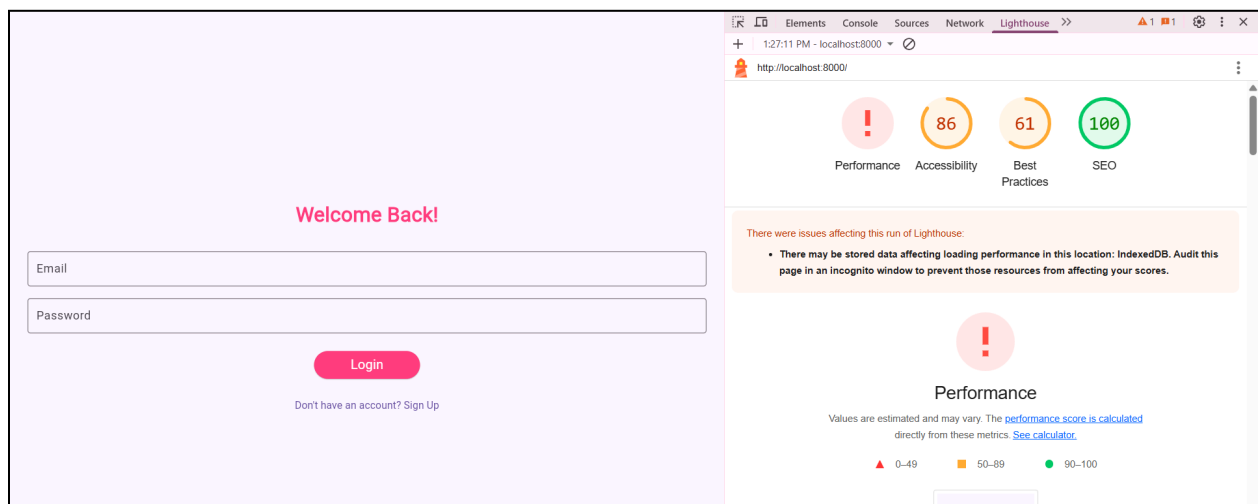
Advantages:

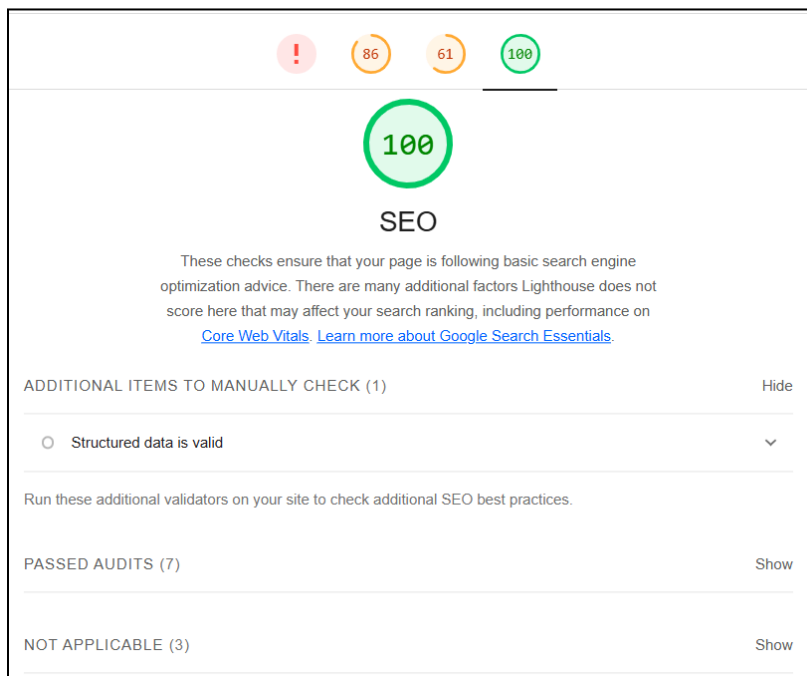
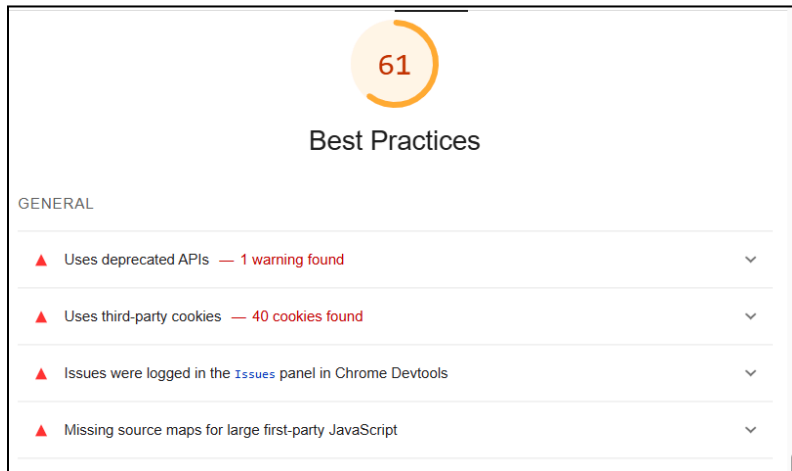
1. **Comprehensive Audit:**
Evaluates performance, accessibility, SEO, and PWA compliance in one report.
2. **Built-in Tool:**
Easily accessible via Chrome DevTools — no extra setup needed.
3. **Actionable Suggestions:**
Provides detailed recommendations to improve app quality.
4. **Free and Open Source:**
No cost involved, and it's maintained by Google with regular updates.
5. **PWA-Specific Checks:**
Focuses on service workers, manifest files, and offline support essential for PWAs.

Disadvantages:

1. **Simulated Environment:**
Tests are run in a simulated environment, which may not reflect real-world usage perfectly.
2. **Limited Offline Test Accuracy:**
Might not catch all edge cases of offline behavior or background sync functionality.
3. **Performance May Vary:**
Different results can appear on different devices or under varying conditions.
4. **Doesn't Replace Manual Testing:**
Can't test UI/UX or real-time functionality thoroughly — manual checks are still needed.
5. **Requires Understanding:**
Interpreting some technical suggestions may be difficult for beginners.

Output:





Conclusion:

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