Vivekanand Education Society's Institute of Technology

An Autonomous Institute Affiliated to University of Mumbai
Hashu Advani Memorial Complex, Collector Colony, Chembur East, Mumbai - 400074.

Department of Information Technology

CERTIFICATE

This is to certify that **Saachi Raheja** of **D15B** semester **VI,** have successfully completed necessary experiments in the **MAD & PWA Lab** under my supervision in **VES Institute of Technology** during the academic year **2024-2025.**

Lab Teacher
Dr. Ravita Mishra
Signature:

Head of Department
Dr. Mrs. Shalu Chopra
Signature:

Principal

Name of the Course : MAD & PWA Lab **Course Code :** ITL604

Year/Sem/Class : D15A/D15B **A.Y.:** 24-25

Faculty Incharge : Dr. Ravita Mishra

Lab Teachers : Dr. Ravita Mishra.

Email : ravita.mishra@ves.ac.in

Programme Outcomes: The graduate will be able to:

PO1) Basic Engineering knowledge: An ability to apply the fundamental knowledge in mathematics, science and engineering to solve problems in Computer engineering.

PO2) Problem Analysis: Identify, formulate, research literature and analyze computer engineering problems reaching substantiated conclusions using first principles of mathematics, natural sciences and computer engineering and sciences.

PO3) Design/ Development of Solutions: Design solutions for complex computer engineering problems and design system components or processes that meet specified needs with appropriate consideration for public health and safety, cultural, societal and environmental considerations.

PO4) Conduct investigations of complex engineering problems using research-based knowledge and research methods including design of experiments, analysis and interpretation of data and synthesis of information to provide valid conclusions.

PO5) Modern Tool Usage: Create, select and apply appropriate techniques, resources and modern computer engineering and IT tools including prediction and modeling to complex engineering activities with an understanding of the limitations.

PO6) The Engineer and Society: Apply reasoning informed by contextual knowledge to assess societal, health, safety, legal and cultural issues and the consequent responsibilities relevant to computer engineering practice.

PO7) Environment and Sustainability: Understand the impact of professional computer engineering solutions in societal and environmental contexts and demonstrate knowledge of and need for sustainable development.

PO8) Ethics: Apply ethical principles and commit to professional ethics and responsibilities and norms of computer engineering practice.

PO9) Individual and Team Work: Function effectively as an individual, and as a member or leader in diverse teams and in multidisciplinary settings.

PO10) Communication: Communicate effectively on complex engineering activities with the engineering community and with society at large, such as being able to comprehend and write effective reports and design documentation, make effective presentations and give and receive clear instructions.

PO11) Project Management and Finance: Demonstrate knowledge and understanding of computer engineering and management principles and apply these to one's own work, as a member and leader in a team, to manage projects and in multidisciplinary environments.

PO12) Life-long Learning: Recognize the need for and have the preparation and ability to engage in independent and lifelong learning in the broadest context of technological change.

Program specific Outcomes

PSO1) An ability to manage and analyze data / information effectively for making better decisions.

PSO2) Demonstrate the ability to use state of the art technologies and tools including Free and Open Source Software (FOSS) tools in developing software.

Lab Objectives:

Sr. No.	Lab Objectives
The Lab experiments aims:	
1	Learn the basics of the Flutter framework.
2	Develop the App UI by incorporating widgets, layouts, gestures and animation
3	Create a production ready Flutter App by including files and firebase backend service.
4	Learn the Essential technologies, and Concepts of PWAs to get started as quickly and efficiently as possible
5	Develop responsive web applications by combining AJAX development techniques with the jQuery JavaScript library.
6	Understand how service workers operate and also learn to Test and Deploy PWA.

Lab Outcomes:

Sr. No.	Lab Outcomes	Cognitive levels of attainment as per Bloom's Taxonomy
On Completion of the course the learner/student should be able to:		
1	Understand cross platform mobile application development using Flutter framework	L1, L2
2	Design and Develop interactive Flutter App by using widgets, layouts, gestures and animation	L3
3	Analyze and Build production ready Flutter App by incorporating backend services and deploying on Android / iOS	L3, L4
4	Understand various PWA frameworks and their requirements	L1, L2
5	Design and Develop a responsive User Interface by applying PWA Design techniques	L3
6	Develop and Analyse PWA Features and deploy it over app hosting solutions	L3, L4

Index

Sr. No	Experiment Title	LO	DOP	DOS	Grade
1.	To install and configure the Flutter Environment	LO1			
2.	To design Flutter UI by including common widgets.	LO2			
3.	To include icons, images, fonts in Flutter app	LO2			
4.	To create an interactive Form using form widget	LO2			
5.	To apply navigation, routing and gestures in Flutter App	LO2			
6.	To Connect Flutter UI with fireBase database	LO3			
7.	To write meta data of your Ecommerce PWA in a Web app manifest file to enable “add to homescreen feature”.	LO4			
8.	To code and register a service worker, and complete the install and activation process for a new service worker for the E-commerce PWA	LO5			
9.	To implement Service worker events like fetch, sync and push for E-commerce PWA	LO5			
10.	To study and implement deployment of Ecommerce PWA to GitHub Pages.	LO5			
11.	To use google Lighthouse PWA Analysis Tool to test the PWA functioning.	LO6			
12.	Assignment-1	LO1,LO2 ,LO3			
13.	Assignment-2	LO4,LO5 ,LO6			

MAD & PWA Lab Journal

Experiment No.	01
Experiment Title.	To install and configure the Flutter Environment
Roll No.	46
Name	SAACHI RAHEJA
Class	D15B
Subject	MAD & PWA Lab
Lab Outcome	LO1: Understand cross platform mobile application development using Flutter framework
Grade:	

EXP 1: Installation and Configuration of Flutter Environment.

Aim: To install and configure the Flutter development environment on your system to start building applications with Flutter.

Theory:Flutter is an open-source framework developed by Google to create cross-platform applications from a single codebase. It enables the development of high-performance apps for Android, iOS, Web, and Desktop. Flutter uses Dart, a programming language developed by Google, which offers fast compilation and powerful async features.

Key Components for Setting Up the Flutter Environment:

1. Flutter SDK:

○ The core development kit for building Flutter applications. It includes the Flutter framework, which provides widgets and tools for creating UIs, and the engine, which handles rendering using the Skia graphics library.

○ It also includes Flutter CLI tools like flutter doctor, flutter run, and flutter build for managing projects and debugging.

2. Dart SDK:

○ Dart is the programming language used by Flutter. It is object-oriented and supports asynchronous programming.

○ Dart uses Just-In-Time (JIT) compilation during development for fast build times and Hot Reload, and Ahead-Of-Time (AOT) compilation for production builds for better performance.

3. IDE (Integrated Development Environment):

○ Android Studio: A full-featured IDE that includes Flutter and Dart plugins, device emulators, and tools for debugging and profiling.

○ Visual Studio Code (VS Code): A lightweight editor with excellent Flutter and Dart plugin support for development and debugging.

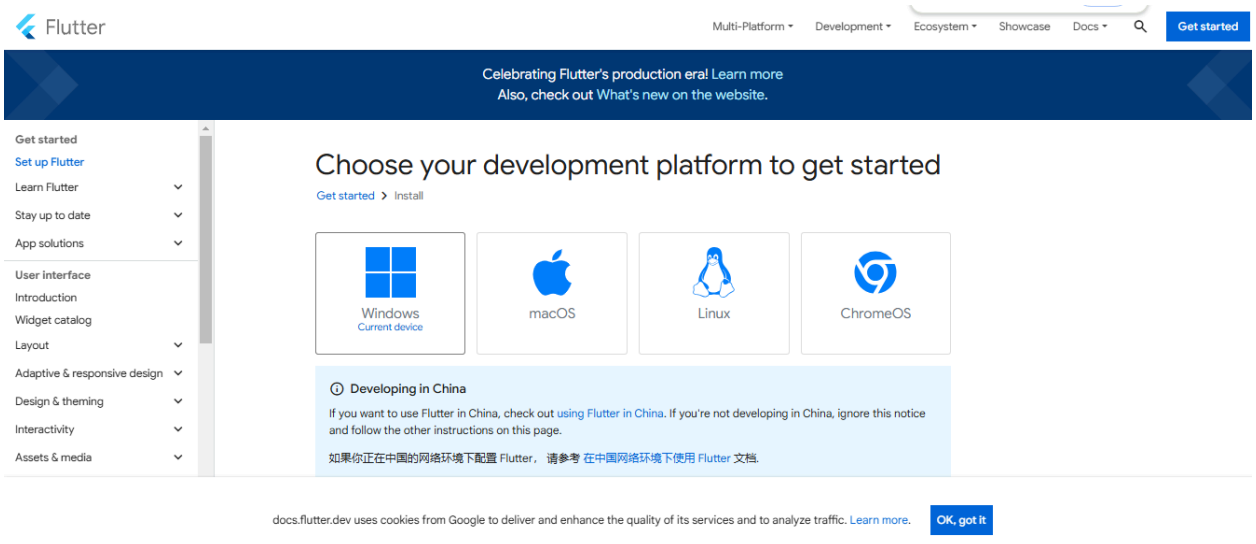
4. Platform-Specific Tools:

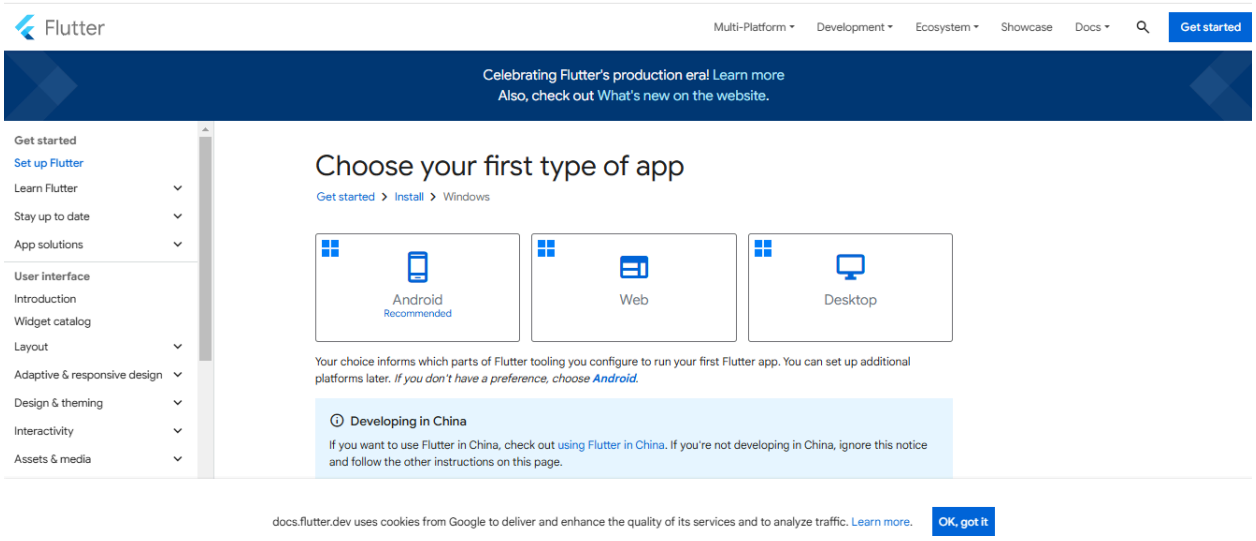
○ Android Studio: Required for Android development with Flutter, including the Android SDK and emulator.

○ Xcode: Needed for iOS development on macOS, with tools like the iOS Simulator

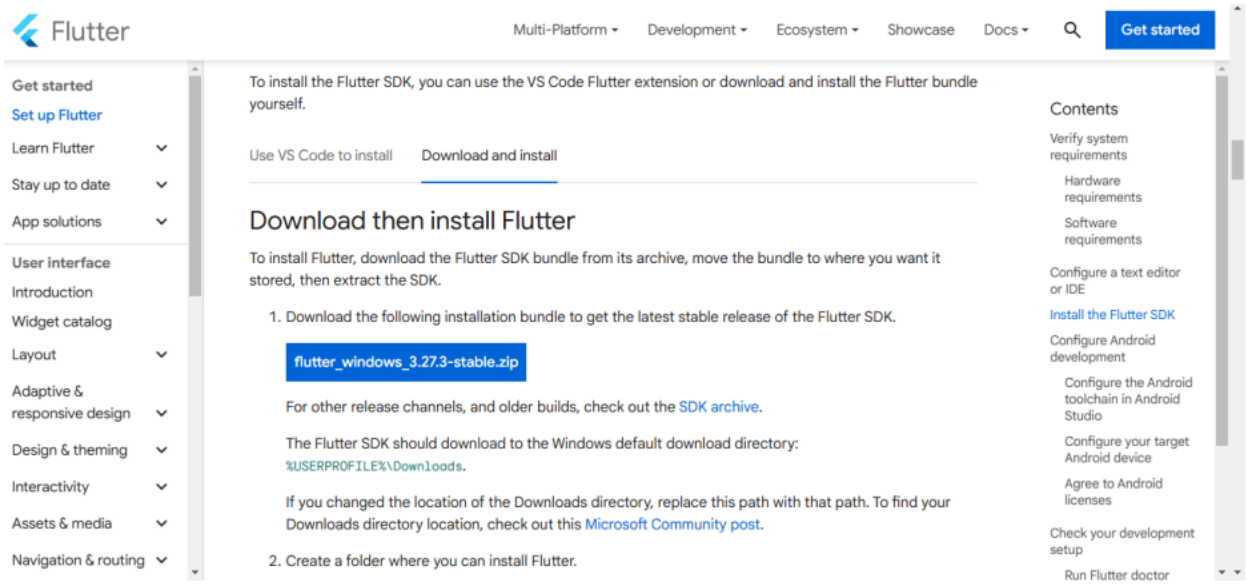
Steps To Install Flutter:

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





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

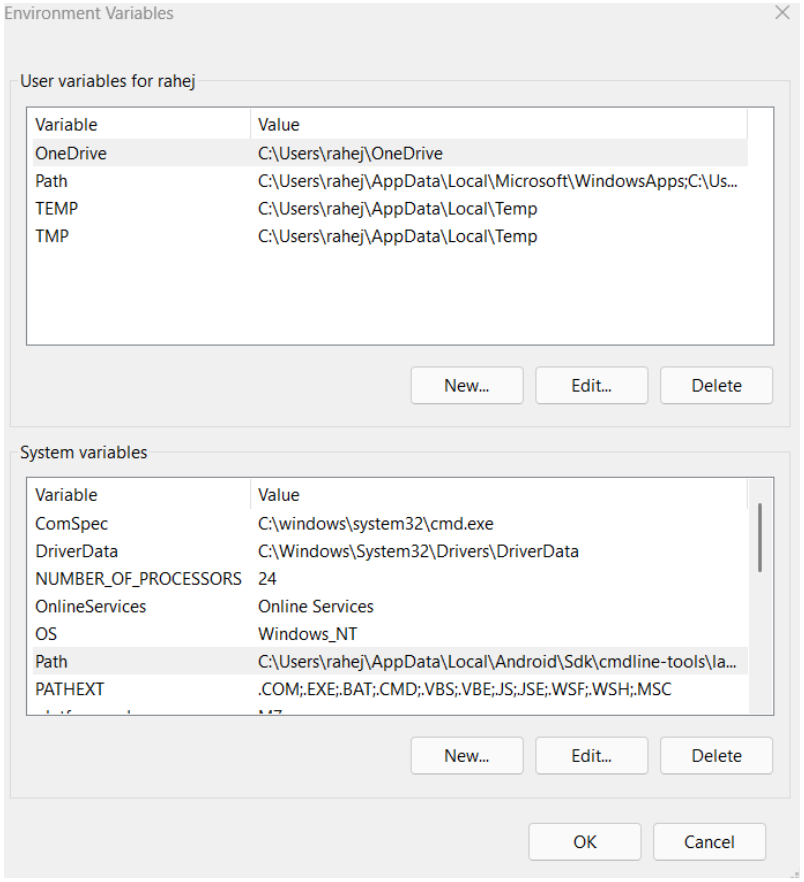


Step 3: When your download is complete, extract the zip file and place it in the desired
installation folder or location, for example, C: /Flutter.

Step 4: To run the Flutter command in regular windows console, you need to update the system

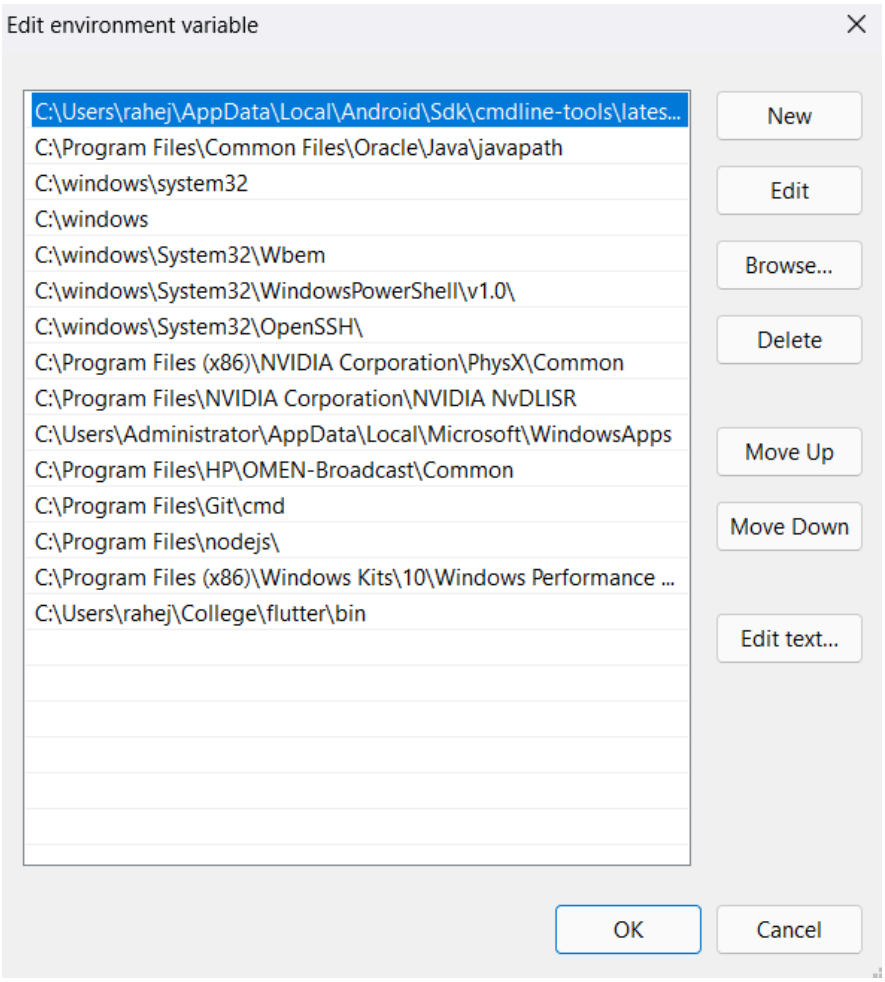
path to include the flutter bin directory. The following steps are required to do this:

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



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

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



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

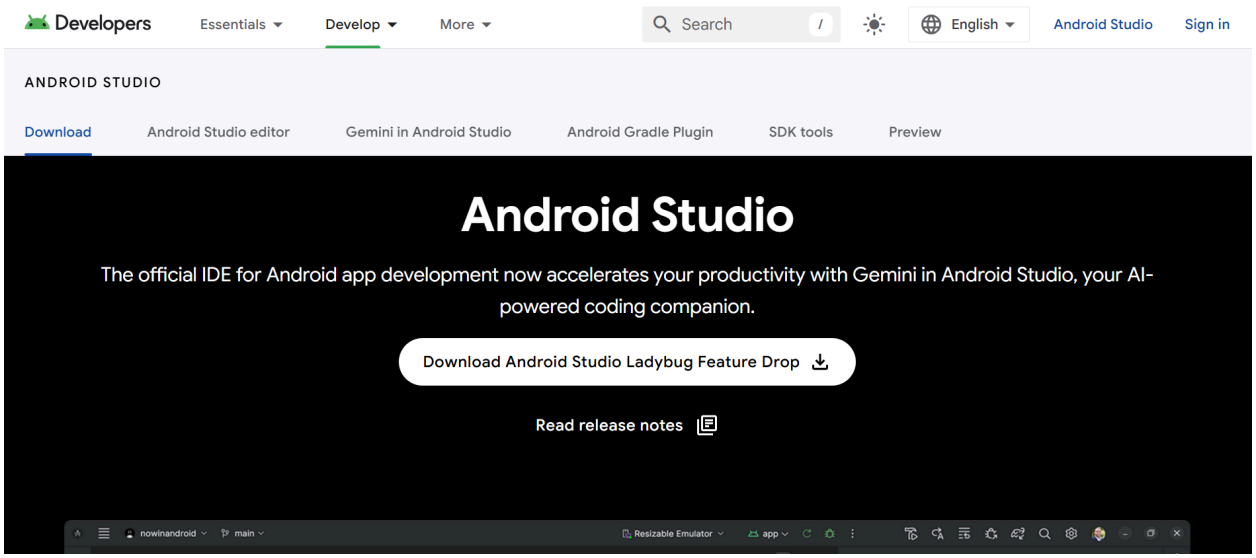
Now, run the $ flutter doctor command. This command checks for all the requirements of Flutter app development and displays a report of the status of your Flutter installation.

```
C:\Users\rahej>flutter
Manage your Flutter app development.

Common commands:

  flutter create <output directory>
    Create a new Flutter project in the specified directory.

  flutter run [options]
    Run your Flutter application on an attached device or in an emulator.

Usage: flutter <command> [arguments]

Global options:
-h, --help                  Print this usage information.
-v, --verbose               Noisy logging, including all shell commands executed.
                            If used with "--help", shows hidden options. If used with "flutter doctor", shows additional
                            diagnostic information. (Use "-vv" to force verbose logging in those cases.)
-d, --device-id             Target device id or name (prefixes allowed).
    --version               Reports the version of this tool.
    --enable-analytics      Enable telemetry reporting each time a flutter or dart command runs.
    --disable-analytics     Disable telemetry reporting each time a flutter or dart command runs, until it is
                            re-enabled.
    --suppress-analytics    Suppress analytics reporting for the current CLI invocation.
```

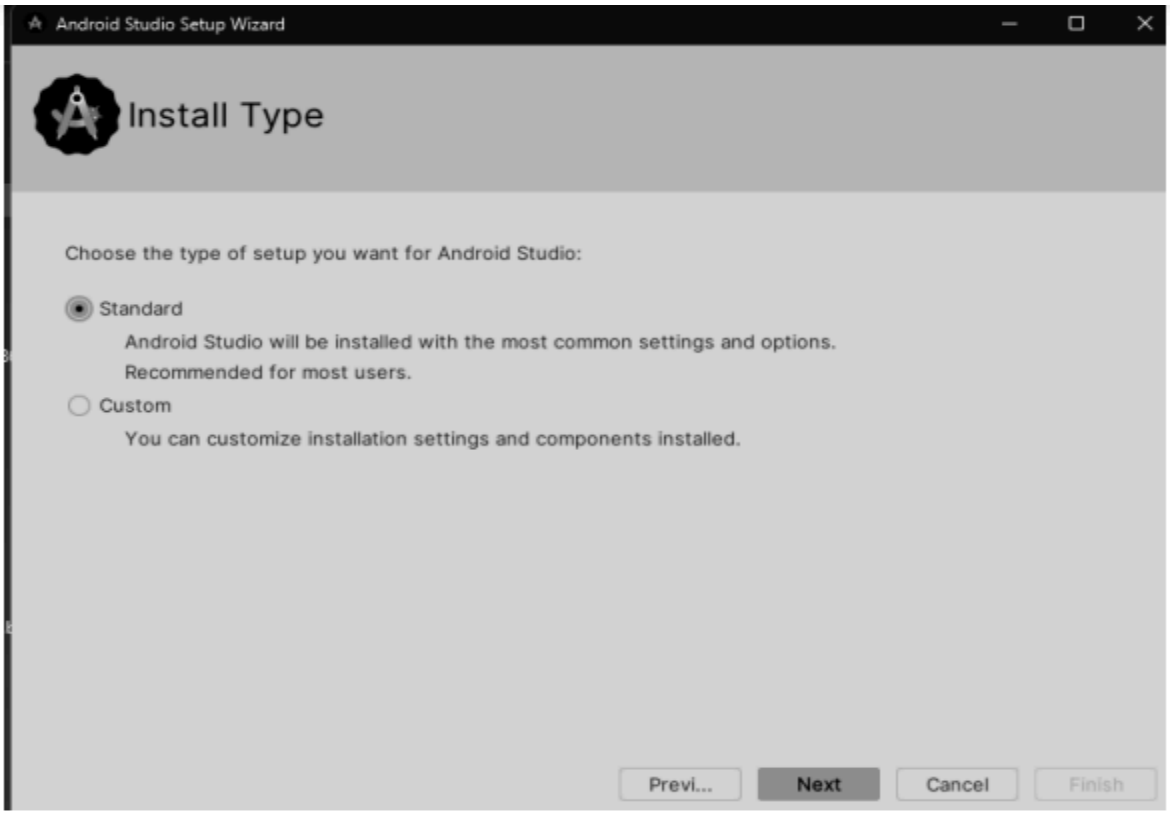
Step 6: When you run the above command, it will analyze the system and show its report, as shown in the below image. Here, you will find the details of all missing tools, which required to run Flutter as well as the development tools that are available but not connected with the device.

```
C:\Users\rahej>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 (Windows 11 or higher, 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 17.13.2)
[✓] Android Studio (version 2024.3)
[✓] VS Code (version 1.98.0)
[✓] Connected device (3 available)
[✓] Network resources

• No issues found!
```

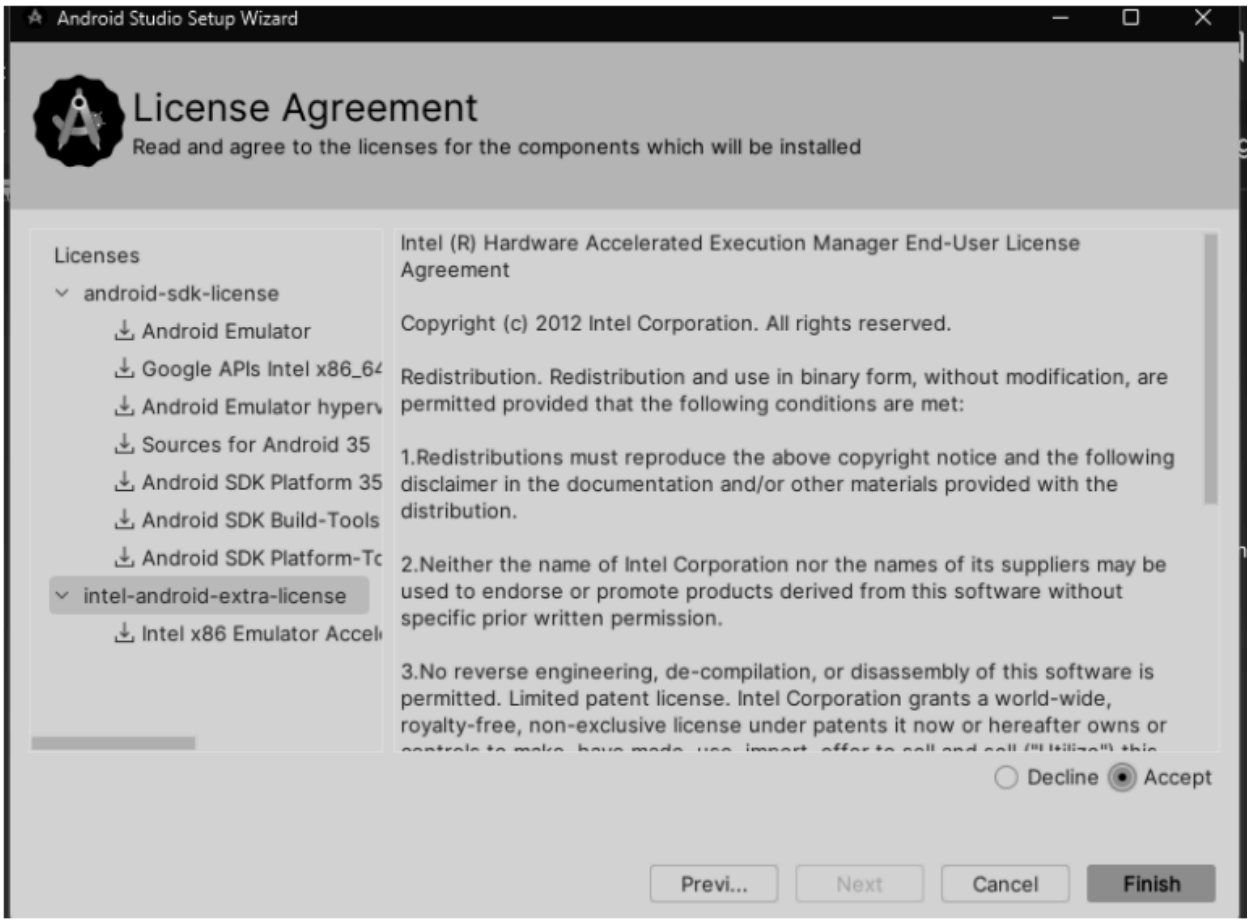
Step 7: Install the Android SDK. If the flutter doctor command does not find the Android SDK tool in your system, then you need first to install the Android Studio IDE. To install Android Studio IDE, do the following steps.



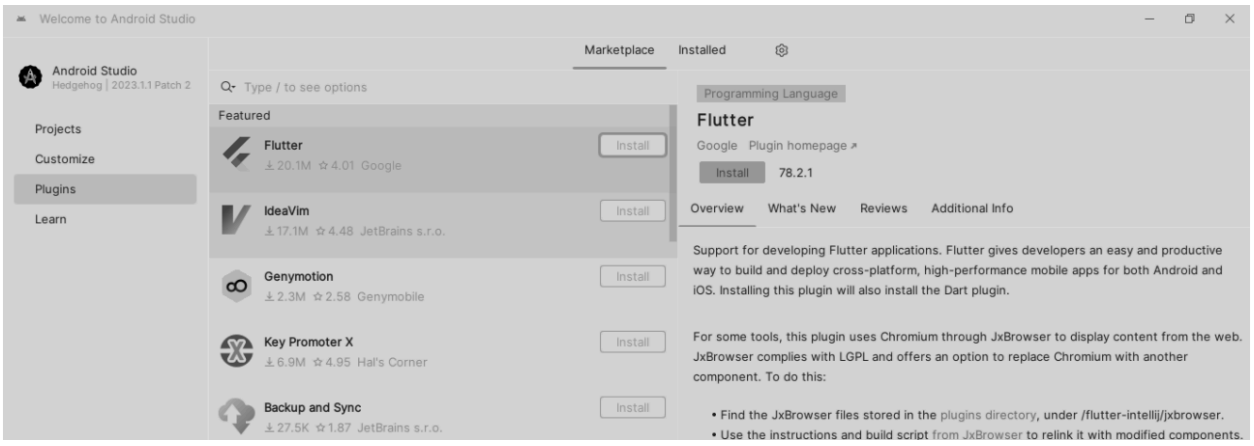
Step 8: After opening the installer you will see the following. Select standard and click next.



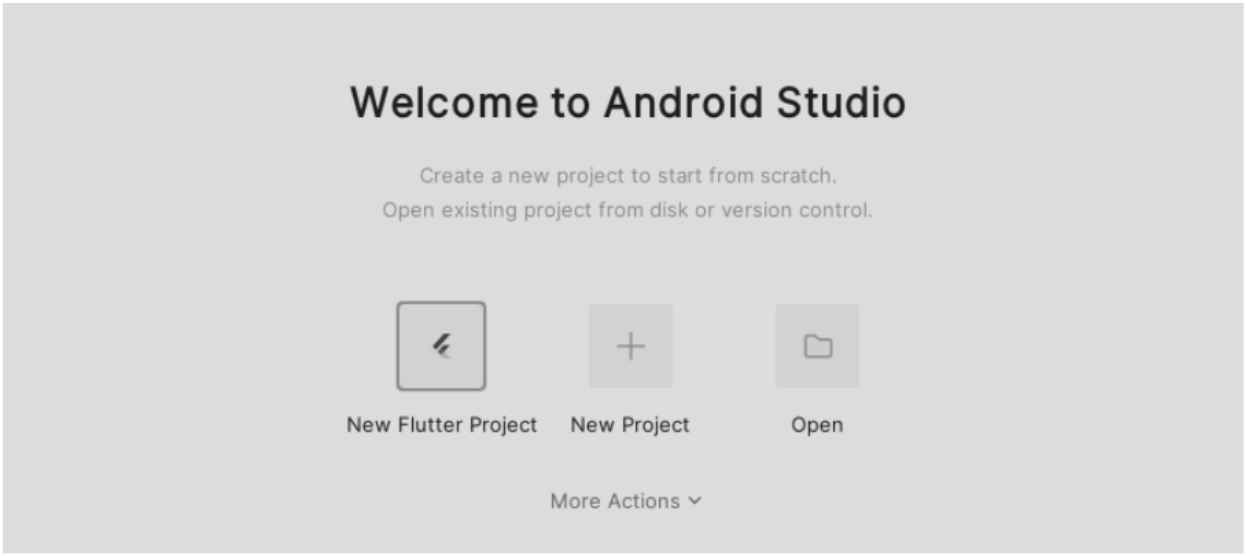
Step 9: Accept all and finish the installation.



Step 10: Install Flutter and Dart plugins from Marketplace.

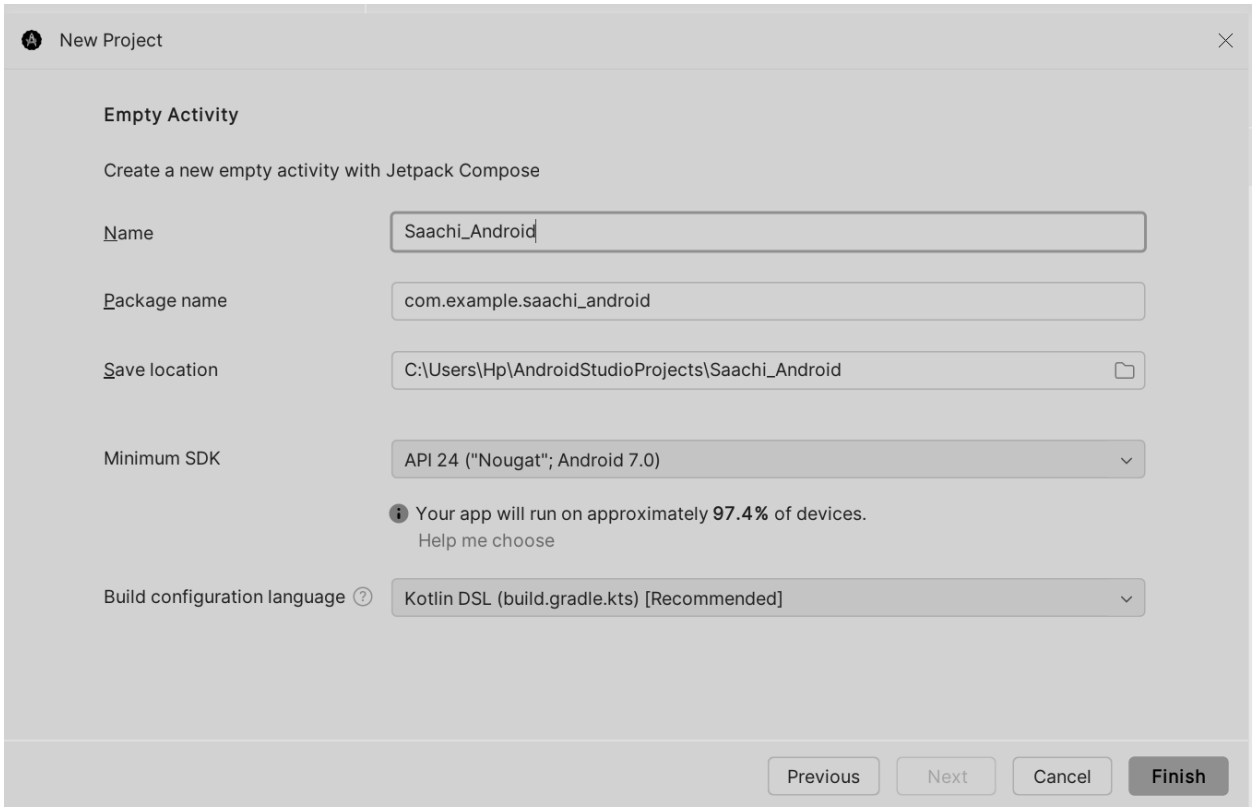


Step 11: Click on New Flutter Project.



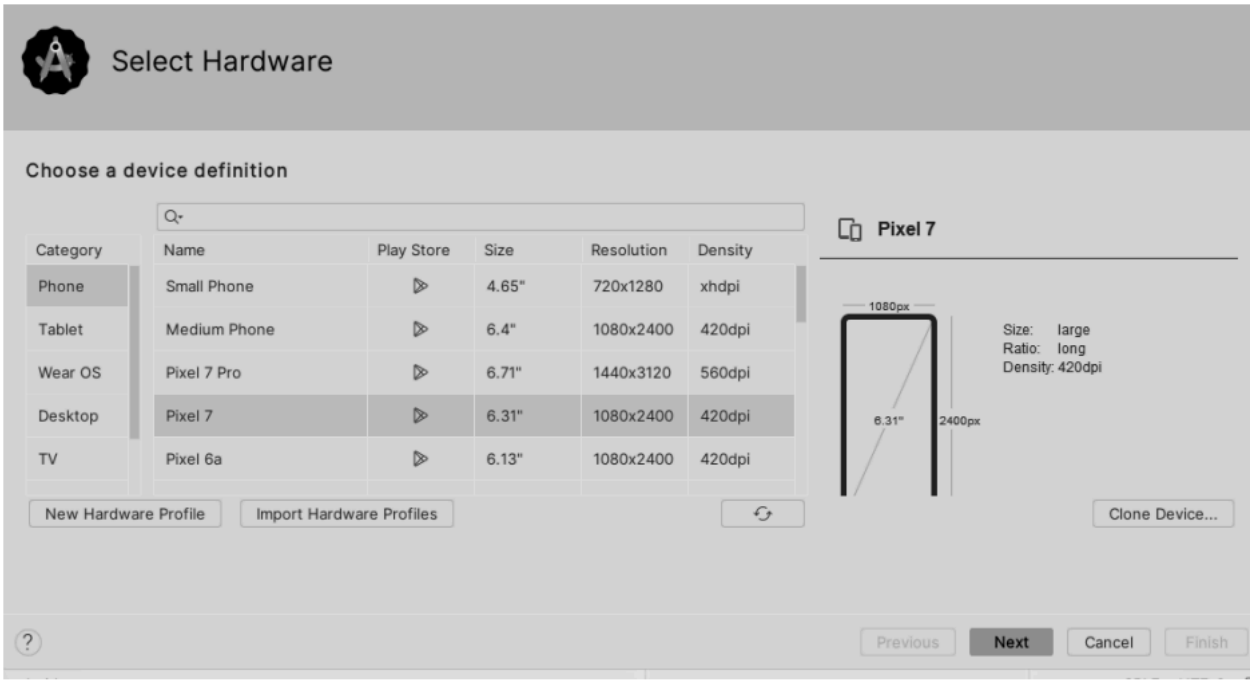
Step 12: Set path C:\flutter\flutter

Step 13: Enter a name for project and click on create.

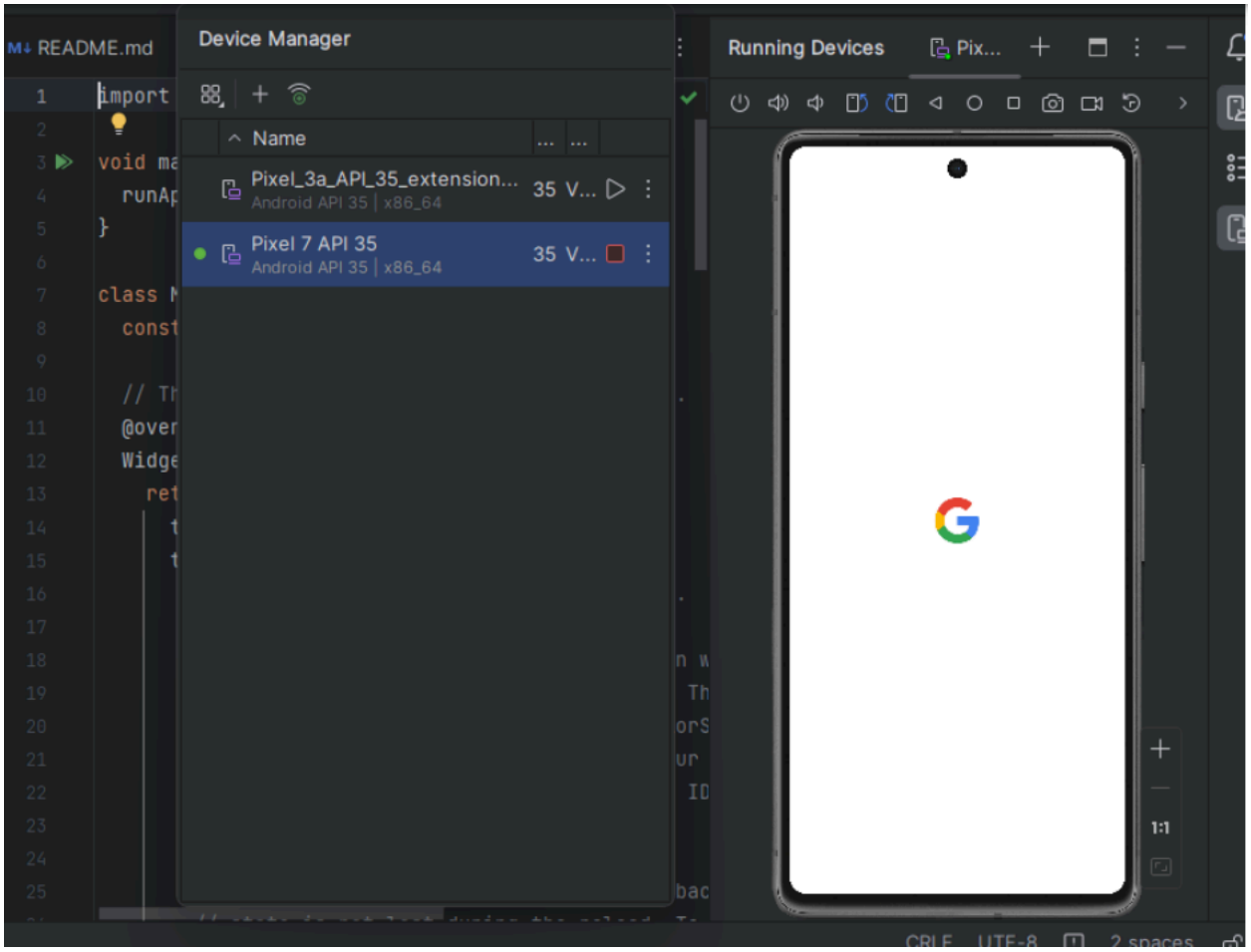


Step 14: Go to Menu > Tools > Device Manager

Step 15: Choose your device on which you want to run your project and click Next.



Step 16 : Click on play button in the toolbar above and you will see the emulator starting. It will take time to load for the first time.



Conclusion:Flutter allows developers to build cross-platform applications with a **single codebase**. The **Flutter SDK**, **Dart programming language**, and the right **IDE** (Android Studio or Visual Studio Code) are essential for setting up the development environment. With tools for fast compilation, real-time changes (Hot Reload), and native-like performance, Flutter makes it easy to create powerful, multi-platform apps.

MAD & PWA Lab Journal

Experiment No.	02
Experiment Title.	To design Flutter UI by including common widgets.
Roll No.	46
Name	SAACHI RAHEJA
Class	D15B
Subject	MAD & PWA Lab
Lab Outcome	LO2: Design and Develop interactive Flutter App by using widgets, layouts, gestures and animation
Grade:	

Aim: To design Flutter UI by including common widgets.

Theory: Widgets are the building blocks of a Flutter application. In Flutter, everything is a widget, from simple elements like text and images to complex structures like entire layouts and navigations. Flutter provides a rich set of predefined widgets to create various UI components, which can be combined to build a complex UI.

Common Flutter Widgets:

1. Text Widget:
   - Used to display text on the screen. You can customize its style, size, color, and alignment.
2. Container Widget:
   - A box that can contain other widgets. It is used for styling, adding padding, margin, alignment, and background color to widgets.
3. Row and Column Widgets:
   - Row: Arranges widgets horizontally.
   - Column: Arranges widgets vertically.
   - These widgets are fundamental for creating flexible layouts and positioning UI elements.
4. Image Widget:
   - Used to display images in the app, either from assets, network, or file system.
5. Button Widgets:
   - Flutter offers several button widgets like RaisedButton, FlatButton, ElevatedButton, and IconButton that are used for interaction. These buttons are essential for handling user input and triggering actions.
6. TextField Widget:
   - Used for user input. It provides an editable field where the user can type text.
7. Scaffold Widget:
   - This is a top-level container that holds the structure of the UI. It includes the app bar, body, drawer, and bottom navigation bar. It provides a standard layout for the app.

8. ListView Widget:
○ A scrolling widget that allows the display of a long list of items. It is used for displaying dynamic content efficiently.

Layouts in Flutter:

- Padding: Adds space around a widget.
- Align: Aligns a widget within its parent.
- Expanded: Makes a widget expand to fill available space in a Row, Column, or Flex.
- Stack: Used for placing widgets on top of one another.

Conclusion: We learned how to design a basic Flutter UI by utilizing common widgets such as Text, Container, Row, Column, Image, Button, TextField, ListView, and Scaffold. These widgets provide a flexible and powerful way to create UIs that are visually appealing and functional. By combining these widgets, you can build complex layouts that cater to your app's design needs.

- Text and Container are fundamental for displaying text and styling.
- Row and Column are essential for structuring layouts, either horizontally or vertically.
- Buttons allow users to interact with the app.
- Scaffold provides a basic structure for the app, including app bars, bodies, and drawers.
- ListView is ideal for displaying lists of items, especially when the content is dynamic or long.

Mastering these common widgets enables developers to design clean and efficient UIs that cater to the needs of modern mobile applications.

Code:

```
import 'package:crimetrack/validation/validator.dart';
import 'package:flutter/material.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:fluttertoast/fluttertoast.dart'; // Import FlutterToast
```

```
import 'verify_page.dart'; // Import VerifyEmailScreen
import '../app_colors.dart'; // Import AppColors

class RegScreen extends StatefulWidget {
  const RegScreen({Key? key}) : super(key: key);

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

class _RegScreenState extends State<RegScreen> {
  final _formKey = GlobalKey<FormState>();
  final TextEditingController _nameController = TextEditingController();
  final TextEditingController _emailController = TextEditingController();
  final TextEditingController _passwordController = TextEditingController();
              final        TextEditingController        _confirmPasswordController        =
TextEditingController();

  bool _isPasswordVisible = false;
  bool _isConfirmPasswordVisible = false;
  bool _isLoading = false;

  FocusNode _nameFocusNode = FocusNode();
  FocusNode _emailFocusNode = FocusNode();
  FocusNode _passwordFocusNode = FocusNode();
  FocusNode _confirmPasswordFocusNode = FocusNode();

  @override
  void dispose() {
    _nameController.dispose();
    _emailController.dispose();
    _passwordController.dispose();
    _confirmPasswordController.dispose();
    _nameFocusNode.dispose();
    _emailFocusNode.dispose();
```

```
  _passwordFocusNode.dispose();
  _confirmPasswordFocusNode.dispose();
  super.dispose();
}

Future<void> _registerUser() async {
  if (_formKey.currentState?.validate() ?? false) {
    setState(() => _isLoading = true);
    try {
      UserCredential userCredential = await FirebaseAuth.instance
          .createUserWithEmailAndPassword(
          email: _emailController.text, password: _passwordController.text);

      // Set display name after user is created
      await userCredential.user?.updateDisplayName(_nameController.text);

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

      // Show success toast
      Fluttertoast.showToast(
        msg: "Registration Successful! Please verify your email.",
        toastLength: Toast.LENGTH_SHORT,
        gravity: ToastGravity.BOTTOM,
        timeInSecForIosWeb: 1,
        backgroundColor: AppColors.successColor,
        textColor: AppColors.textColor,
        fontSize: 16.0,
      );

      // Navigate to VerifyEmailScreen
      Navigator.pushReplacement(
        context,
        MaterialPageRoute(builder: (context) => VerifyEmailScreen()),
      );
```

```
    } catch (e) {
      // Show error toast
      Fluttertoast.showToast(
        msg: "Error: ${e.toString()}",
        toastLength: Toast.LENGTH_SHORT,
        gravity: ToastGravity.BOTTOM,
        timeInSecForIosWeb: 1,
        backgroundColor: AppColors.errorColor,
        textColor: AppColors.textColor,
        fontSize: 16.0,
      );
    } finally {
      setState(() => _isLoading = false);
    }
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Stack(
      children: [
        // Background Gradient
        Container(
          height: double.infinity,
          width: double.infinity,
          decoration: const BoxDecoration(
            gradient: LinearGradient(
              colors: [AppColors.primaryColor, AppColors.secondaryColor],
            ),
          ),
          child: const Padding(
            padding: EdgeInsets.only(top: 60.0, left: 22),
            child: Text(
              'Create Your\nAccount',
```

```
                    style: TextStyle(fontSize: 30, color: Colors.white, fontWeight: FontWeight.bold),
                ),
              ),
            ),
            Padding(
              padding: const EdgeInsets.only(top: 200.0),
              child: Container(
                decoration: const BoxDecoration(
                  borderRadius: BorderRadius.only(topLeft: Radius.circular(40), topRight: Radius.circular(40)),
                  color: AppColors.backgroundColor,
                ),
                height: double.infinity,
                width: double.infinity,
                child: SingleChildScrollView(
                  padding: const EdgeInsets.symmetric(horizontal: 18.0, vertical: 30),
                  child: Form(
                    key: _formKey,
                    child: Column(
                      children: [
                                  _buildTextField('Full Name', _nameController, false, Validator.validateName, _nameFocusNode),
                        const SizedBox(height: 10),
                                      _buildTextField('Email', _emailController, false, Validator.validateEmail, _emailFocusNode),
                        const SizedBox(height: 10),
                                  _buildTextField('Password', _passwordController, true, Validator.validatePassword, _passwordFocusNode),
                        const SizedBox(height: 10),
                            _buildTextField('Confirm Password', _confirmPasswordController, true, (value) {
                                  return Validator.validateConfirmPassword(value ?? '', _passwordController.text);
                        }, _confirmPasswordFocusNode),
```

```
                const SizedBox(height: 50),
                GestureDetector(
                  onTap: _isLoading ? null : _registerUser,
                  child: Container(
                    height: 55,
                    width: 300,
                    decoration: BoxDecoration(
                      borderRadius: BorderRadius.circular(30),
                      gradient: const LinearGradient(
                        colors: [AppColors.primaryColor, AppColors.secondaryColor],
                      ),
                    ),
                    child: Center(
                      child: _isLoading
                          ? const CircularProgressIndicator(color: Colors.white)
                          : const Text(
                        'SIGN UP',
                              style: TextStyle(fontWeight: FontWeight.bold, fontSize: 20,
color: AppColors.buttonTextColor),
                      ),
                    ),
                  ),
                ),
                const SizedBox(height: 50),
              ],
            ),
          ),
        ),
      ),
    ),
  );
 }
```

```
  // Updated text field method
    Widget _buildTextField(String label, TextEditingController controller, bool isPassword, String? Function(String?) validator, FocusNode focusNode) {
   return TextFormField(
     controller: controller,
     focusNode: focusNode, // Assign focus node
                        obscureText: isPassword ? (!_isPasswordVisible && !_isConfirmPasswordVisible) : false,
     cursorColor: AppColors.primaryColor, // Set cursor color to primary
    // Set selection color to primary
     decoration: InputDecoration(
       labelText: label,
       labelStyle: TextStyle(
         fontWeight: FontWeight.bold,
                           color: focusNode.hasFocus ? AppColors.primaryColor : AppColors.secondaryColor, // Change label color on focus
       ),
       suffixIcon: isPassword
           ? IconButton(
         icon: Icon(
                     label == 'Password' ? (_isPasswordVisible ? Icons.visibility : Icons.visibility_off)
               : (_isConfirmPasswordVisible ? Icons.visibility : Icons.visibility_off),
           color: Colors.grey,
         ),
         onPressed: () {
           setState(() {
             if (label == 'Password') {
               _isPasswordVisible = !_isPasswordVisible;
             } else {
               _isConfirmPasswordVisible = !_isConfirmPasswordVisible;
             }
           });
         },
       )
```

```
          : null,
      ),
      validator: validator,
          style: TextStyle(color: focusNode.hasFocus ? AppColors.primaryColor :
AppColors.textColor), // Text color on focus
    );
  }
}
```

Login Screen:

```
import 'package:flutter/material.dart';
import 'package:firebase_auth/firebase_auth.dart'; // Import FirebaseAuth package
import 'package:crimetrack/screens/forgot_password.dart'; // Import
ForgotPasswordScreen
import 'package:crimetrack/screens/home_screen.dart'; // Import HomeScreen
import 'package:crimetrack/screens/register_screen.dart'; // Import RegisterScreen
import 'package:crimetrack/validation/validator.dart'; // Import Validator class
import '../app_colors.dart'; // Import AppColors class
import 'package:fluttertoast/fluttertoast.dart'; // Import fluttertoast

class LoginScreen extends StatefulWidget {
  const LoginScreen({Key? key}) : super(key: key);

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

class _LoginScreenState extends State<LoginScreen> {
  final _formKey = GlobalKey<FormState>(); // Key to identify the form
  final TextEditingController _emailController = TextEditingController();
  final TextEditingController _passwordController = TextEditingController();

  bool _isPasswordVisible = false; // Boolean variable to track password visibility
  bool _isLoading = false; // Boolean to track loading state
```

```
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        leading: IconButton(
          icon: const Icon(Icons.arrow_back),
          onPressed: () {
            Navigator.pop(context);
          },
          color: AppColors.backgroundColor, // Set the back button color to white
        ),
              backgroundColor: AppColors.primaryColor, // Use primaryColor from
AppColors
        elevation: 0,
      ),
      body: Stack(
        children: [
          // Background gradient
          Container(
            height: double.infinity,
            width: double.infinity,
            decoration: BoxDecoration(
              gradient: LinearGradient(
                colors: [
                  AppColors.primaryColor, // Dark Blue from AppColors
                  AppColors.secondaryColor, // Light Blue from AppColors
                ],
              ),
            ),
            child: const Padding(
              padding: EdgeInsets.only(top: 60.0, left: 22),
              child: Text(
                'Hello\nSign in!',
                style: TextStyle(
                  fontSize: 30,
```

```
              color: AppColors.backgroundColor, // White color from AppColors
              fontWeight: FontWeight.bold,
            ),
          ),
        ),
      ),
      // Login form container
      Padding(
        padding: const EdgeInsets.only(top: 200.0),
        child: Container(
          decoration: BoxDecoration(
            borderRadius: const BorderRadius.only(
              topLeft: Radius.circular(40),
              topRight: Radius.circular(40),
            ),
              color: AppColors.backgroundColor, // White background for the login
form
          ),
          height: double.infinity,
          width: double.infinity,
          child: Padding(
            padding: const EdgeInsets.only(left: 18.0, right: 18),
            child: Form(
              key: _formKey, // Attach the form key
              child: Column(
                mainAxisAlignment: MainAxisAlignment.center,
                children: [
                  // Email input field with validation
                  TextFormField(
                    controller: _emailController,
                    decoration: InputDecoration(
                      suffixIcon: const Icon(
                        Icons.check,
                        color: Colors.grey,
                      ),
```

```
                  label: Text(
                   'Gmail',
                   style: TextStyle(
                    fontWeight: FontWeight.bold,
                          color: AppColors.secondaryColor, // Set the label color to
secondary color
                   ),
                  ),
                  enabledBorder: UnderlineInputBorder(
                       borderSide: BorderSide(color: AppColors.primaryColor), // Set
the border color to primary color
                  ),
                  focusedBorder: UnderlineInputBorder(
                       borderSide: BorderSide(color: AppColors.primaryColor), // Set
the border color to primary color when focused
                  ),
                  hintStyle: TextStyle(
                          color: AppColors.primaryColor, // Set the hint text color to
primary color
                  ),
                 ),
                 style: TextStyle(
                         color: AppColors.primaryColor, // Set the input text color to
primary color
                 ),
                 validator: (value) {
                      return Validator.validateEmail(value); // Using Validator class to
validate email
                 },
                ),
                const SizedBox(height: 20),
                // Password input field with validation and visibility toggle
                TextFormField(
                 controller: _passwordController,
```

```
                    obscureText: !_isPasswordVisible, // Toggle the visibility based on the boolean value
                  decoration: InputDecoration(
                   suffixIcon: IconButton(
                    icon: Icon(
                     _isPasswordVisible
                         ? Icons.visibility
                         : Icons.visibility_off,
                     color: AppColors.primaryColor,
                    ),
                    onPressed: () {
                     setState(() {
                       _isPasswordVisible = !_isPasswordVisible; // Toggle password visibility
                     });
                    },
                   ),
                   label: Text(
                    'Password',
                    style: TextStyle(
                     fontWeight: FontWeight.bold,
                           color: AppColors.secondaryColor, // Set the label color to secondary color
                    ),
                   ),
                   enabledBorder: UnderlineInputBorder(
                       borderSide: BorderSide(color: AppColors.primaryColor), // Set the border color to primary color
                   ),
                   focusedBorder: UnderlineInputBorder(
                       borderSide: BorderSide(color: AppColors.primaryColor), // Set the border color to primary color when focused
                   ),
                   hintStyle: TextStyle(
```

```
                        color: AppColors.primaryColor, // Set the hint text color to primary color
                    ),
                  ),
                  style: TextStyle(
                        color: AppColors.primaryColor, // Set the input text color to primary color
                  ),
                  validator: (value) {
                     return Validator.validatePassword(value); // Using Validator class to validate password
                  },
                ),
                const SizedBox(height: 20),
                // Forgot password text
                Align(
                  alignment: Alignment.centerRight,
                  child: GestureDetector(
                    onTap: () {
                      // Navigate to ForgotPasswordScreen when tapped
                      Navigator.push(
                        context,
                        MaterialPageRoute(
                          builder: (context) => const ForgotPasswordScreen(),
                        ),
                      );
                    },
                    child: const Text(
                      'Forgot Password?',
                      style: TextStyle(
                        fontWeight: FontWeight.bold,
                        fontSize: 17,
                        color: AppColors.secondaryColor, // Darker color for the text
                      ),
                    ),
```

```
                ),
              ),
              const SizedBox(height: 70),
              // Sign In button with loading indicator
              GestureDetector(
                onTap: () async {
                  if (_formKey.currentState!.validate()) {
                    setState(() {
                      _isLoading = true; // Set loading state to true
                    });

                    // Attempt login using Firebase Authentication
                    try {
                      UserCredential userCredential = await FirebaseAuth.instance
                          .signInWithEmailAndPassword(
                        email: _emailController.text,
                        password: _passwordController.text,
                      );

                      // Check if the email is verified
                                                  if (userCredential.user != null &&
userCredential.user!.emailVerified) {
                        // Show success toast
                        Fluttertoast.showToast(
                          msg: "Login Successful",  // Toast message
                          toastLength: Toast.LENGTH_SHORT,
                          gravity: ToastGravity.BOTTOM,
                          timeInSecForIosWeb: 1,
                          backgroundColor: Colors.green,
                          textColor: Colors.white,
                          fontSize: 16.0,
                        );

                        Navigator.push(
                          context,
```

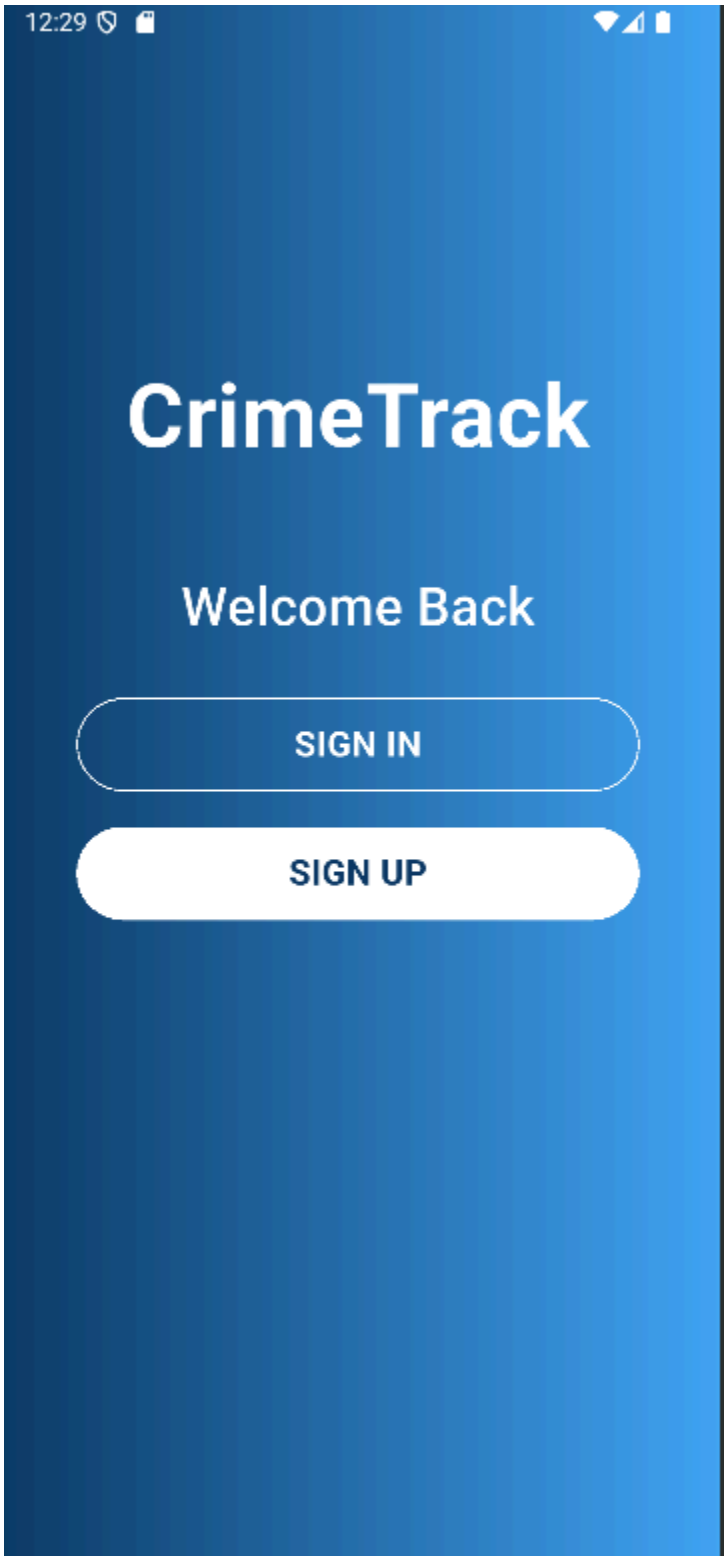
```
      MaterialPageRoute(
        builder: (context) => const HomeScreen(),
      ),
    );
  } else {
    // Show email verification message as toast
    Fluttertoast.showToast(
      msg: "Please verify your email before logging in.",
      toastLength: Toast.LENGTH_SHORT,
      gravity: ToastGravity.BOTTOM,
      timeInSecForIosWeb: 1,
      backgroundColor: Colors.orange,
      textColor: Colors.white,
      fontSize: 16.0,
    );
  }
} catch (e) {
  String errorMessage = 'Error: Invalid Credentials';

  // Handle specific Firebase errors
  if (e is FirebaseAuthException) {
    if (e.code == 'user-not-found') {
      errorMessage = 'No user found for that email.';
    } else if (e.code == 'wrong-password') {
      errorMessage = 'Incorrect password.';
    } else if (e.code == 'invalid-email') {
      errorMessage = 'Invalid email address.';
    }
  }

  // Show error toast
  Fluttertoast.showToast(
    msg: errorMessage,  // Display the error message in toast
    toastLength: Toast.LENGTH_SHORT,
    gravity: ToastGravity.BOTTOM,
```

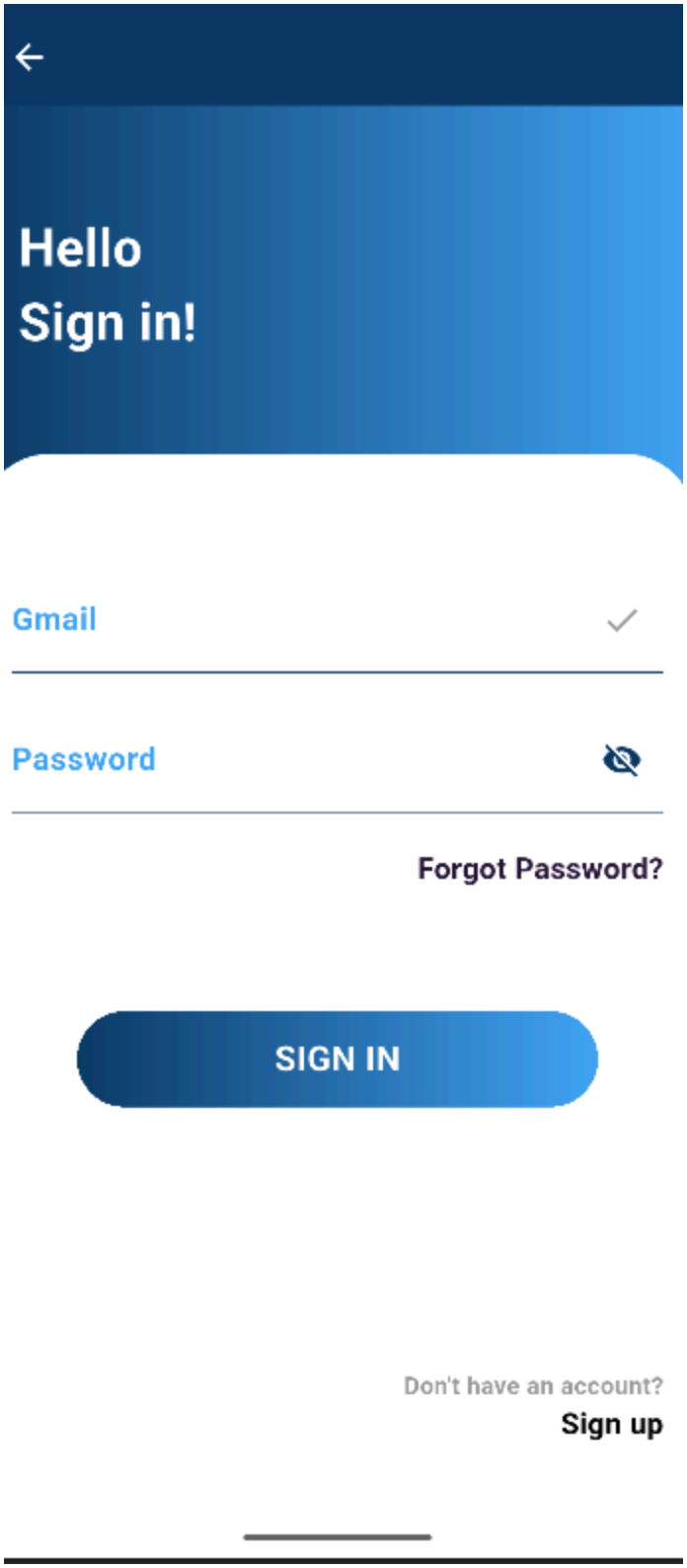
```
                    timeInSecForIosWeb: 1,
                    backgroundColor: Colors.red,
                    textColor: Colors.white,
                    fontSize: 16.0,
                  );
                } finally {
                  setState(() {
                    _isLoading = false; // Set loading state to false
                  });
                }
              }
            },
            child: _isLoading
                ? const CircularProgressIndicator() // Show loading indicator
while processing
                : Container(
                    height: 55,
                    width: 300,
                    decoration: BoxDecoration(
                      borderRadius: BorderRadius.circular(30),
                      gradient: LinearGradient(
                        colors: [
                          AppColors.primaryColor,
                          AppColors.secondaryColor,
                        ],
                      ),
                    ),
                    child: const Center(
                      child: Text(
                        'SIGN IN',
                        style: TextStyle(
                          fontWeight: FontWeight.bold,
                          fontSize: 20,
                          color: AppColors.backgroundColor,
                        ),
```

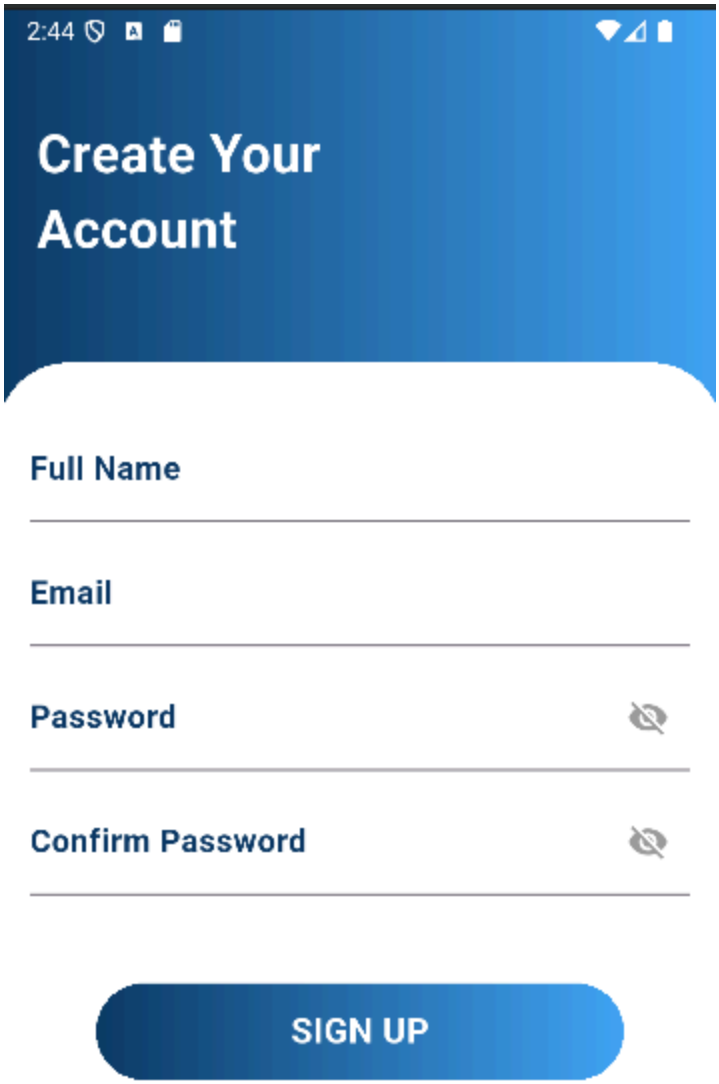
```
                ),
              ),
            ),
          ),
          const SizedBox(height: 150),
          // Sign up link
          Align(
            alignment: Alignment.bottomRight,
            child: Column(
              crossAxisAlignment: CrossAxisAlignment.end,
              children: [
                const Text(
                  "Don't have an account?",
                  style: TextStyle(
                    fontWeight: FontWeight.bold,
                    color: Colors.grey,
                  ),
                ),
                GestureDetector(
                  onTap: () {
                    // Navigate to the Register screen
                    Navigator.push(
                      context,
                      MaterialPageRoute(
                            builder: (context) => const RegScreen(), // Navigate to
RegisterScreen
                      ),
                    );
                  },
                  child: const Text(
                    "Sign up",
                    style: TextStyle(
                      fontWeight: FontWeight.bold,
                      fontSize: 17,
                      color: AppColors.primaryColor, // Black color for "Sign up"
```

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

Output:




Hello
Sign in!
Gmail
Password
Forgot Password?
SIGN IN
Don't have an account?
Sign up


2:44
Create Your Account
Full Name
Email
Password
Confirm Password
SIGN UP

MAD & PWA Lab Journal

Experiment No.	03
Experiment Title.	To include icons, images, fonts in Flutter app
Roll No.	46
Name	SAACHI RAHEJA
Class	D15B
Subject	MAD & PWA Lab
Lab Outcome	LO2: Design and Develop interactive Flutter App by using widgets, layouts, gestures and animation
Grade:	

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

Theory :

1. Icons in Flutter

Icons are essential elements in modern mobile apps as they visually represent actions, features, or categories. In Flutter, icons are used to provide an intuitive and clean user interface.

● Material Icons: Flutter comes with a built-in set of Material Design icons that can be used in any app. These icons are simple and designed to work well across various devices and screen sizes.

● Custom Icons: You can also integrate your own set of icons into the app. These custom icons can be added as image assets and used where required in the UI.

Benefits of Using Icons:

● Enhance usability by providing visual cues.

● Improve app aesthetics by creating a cohesive and modern look.

● Consistent and recognizable symbols for common actions (e.g., home, search, settings).

2. Images in Flutter

Images help to improve the look and feel of an app, offering visual appeal. In Flutter, you can display images from different sources such as local assets, network URLs, or even files stored on the device.

● Asset Images: These are images stored locally within the app's project directory. They can be bundled with the app and accessed efficiently.
● Network Images: Images fetched from the internet, typically hosted on a server or cloud service. Network images provide flexibility to display dynamic content.

● File Images: Images that reside on the device's local storage, often from user uploads or device-specific data.

Benefits of Using Images:

● Add context or visual interest to the app's UI.

● Help users understand the app's content or features quickly.

● Allow for brand identity by displaying logos, banners, or promotional graphics.

3. Fonts in Flutter

Fonts are key to the visual identity of any app, and Flutter allows you to use both default fonts as well as custom fonts to match your app's theme or brand.

● Custom Fonts: You can add fonts in various formats (e.g., TrueType Font .ttf or OpenType Font .otf) and define them in the app's configuration. This allows you to maintain consistency with your brand's typography.

● Default Fonts: Flutter also provides a set of default fonts which are used if no custom fonts are specified.

Benefits of Using Custom Fonts:

■ Improve the app's aesthetic by using unique typography.

■ Maintain brand consistency by incorporating brand-specific font

● Allow for flexibility in styling text elements (headings, body text, etc.) throughout the app.

Code:

```
import 'package:crimetrack/validation/validator.dart';
import 'package:flutter/material.dart';
```

```
import 'package:firebase_auth/firebase_auth.dart';
import 'package:fluttertoast/fluttertoast.dart'; // Import FlutterToast
import 'verify_page.dart'; // Import VerifyEmailScreen
import '../app_colors.dart'; // Import AppColors

class RegScreen extends StatefulWidget {
  const RegScreen({Key? key}) : super(key: key);

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

class _RegScreenState extends State<RegScreen> {
  final _formKey = GlobalKey<FormState>();
  final TextEditingController _nameController = TextEditingController();
  final TextEditingController _emailController = TextEditingController();
  final TextEditingController _passwordController = TextEditingController();
              final        TextEditingController        _confirmPasswordController        =
TextEditingController();

  bool _isPasswordVisible = false;
  bool _isConfirmPasswordVisible = false;
  bool _isLoading = false;

  FocusNode _nameFocusNode = FocusNode();
  FocusNode _emailFocusNode = FocusNode();
  FocusNode _passwordFocusNode = FocusNode();
  FocusNode _confirmPasswordFocusNode = FocusNode();

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

```
  _nameFocusNode.dispose();
  _emailFocusNode.dispose();
  _passwordFocusNode.dispose();
  _confirmPasswordFocusNode.dispose();
  super.dispose();
}

Future<void> _registerUser() async {
  if (_formKey.currentState?.validate() ?? false) {
    setState(() => _isLoading = true);
    try {
      UserCredential userCredential = await FirebaseAuth.instance
          .createUserWithEmailAndPassword(
          email: _emailController.text, password: _passwordController.text);

      // Set display name after user is created
      await userCredential.user?.updateDisplayName(_nameController.text);

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

      // Show success toast
      Fluttertoast.showToast(
        msg: "Registration Successful! Please verify your email.",
        toastLength: Toast.LENGTH_SHORT,
        gravity: ToastGravity.BOTTOM,
        timeInSecForIosWeb: 1,
        backgroundColor: AppColors.successColor,
        textColor: AppColors.textColor,
        fontSize: 16.0,
      );

      // Navigate to VerifyEmailScreen
      Navigator.pushReplacement(
        context,
```

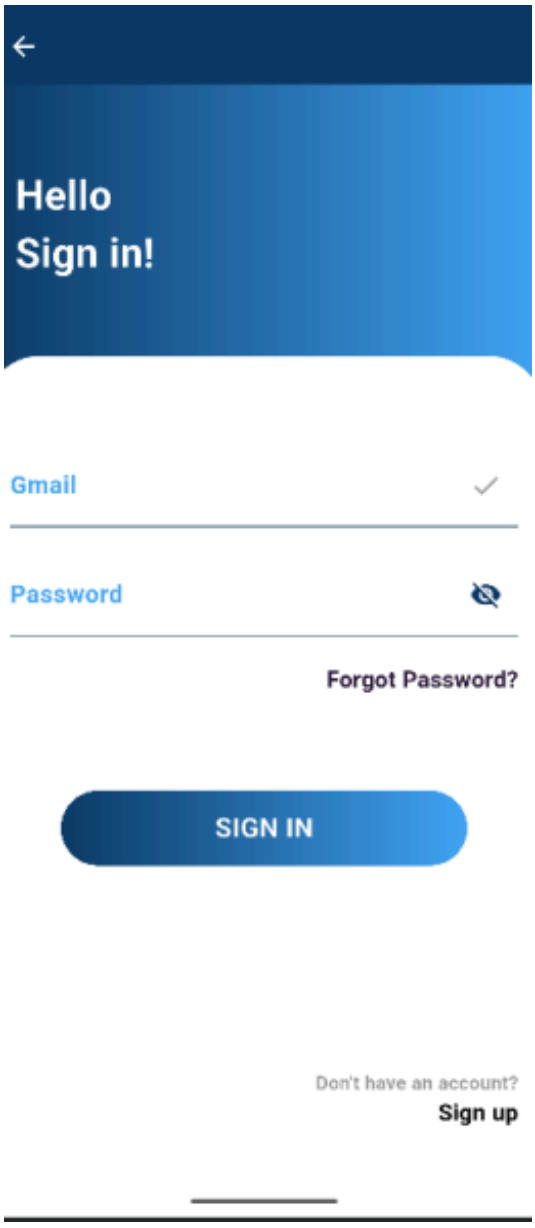
```
        MaterialPageRoute(builder: (context) => VerifyEmailScreen()),
      );
    } catch (e) {
      // Show error toast
      Fluttertoast.showToast(
        msg: "Error: ${e.toString()}",
        toastLength: Toast.LENGTH_SHORT,
        gravity: ToastGravity.BOTTOM,
        timeInSecForIosWeb: 1,
        backgroundColor: AppColors.errorColor,
        textColor: AppColors.textColor,
        fontSize: 16.0,
      );
    } finally {
      setState(() => _isLoading = false);
    }
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Stack(
      children: [
        // Background Gradient
        Container(
          height: double.infinity,
          width: double.infinity,
          decoration: const BoxDecoration(
            gradient: LinearGradient(
              colors: [AppColors.primaryColor, AppColors.secondaryColor],
            ),
          ),
          child: const Padding(
            padding: EdgeInsets.only(top: 60.0, left: 22),
```

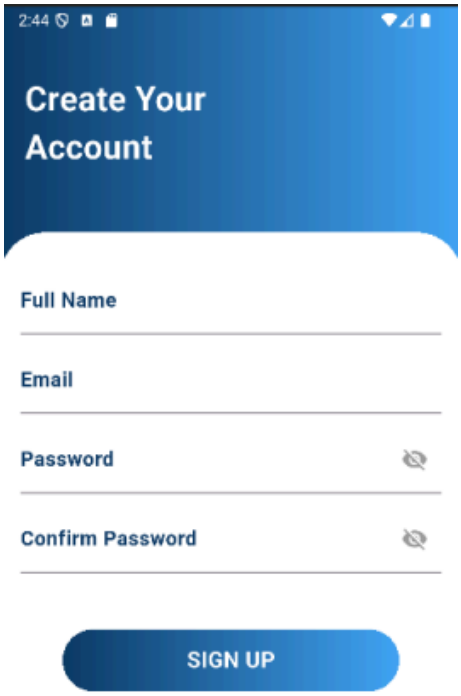
```
          child: Text(
            'Create Your\nAccount',
                    style: TextStyle(fontSize: 30, color: Colors.white, fontWeight:
FontWeight.bold),
          ),
        ),
      ),
      Padding(
        padding: const EdgeInsets.only(top: 200.0),
        child: Container(
          decoration: const BoxDecoration(
            borderRadius: BorderRadius.only(topLeft: Radius.circular(40), topRight:
Radius.circular(40)),
            color: AppColors.backgroundColor,
          ),
          height: double.infinity,
          width: double.infinity,
          child: SingleChildScrollView(
            padding: const EdgeInsets.symmetric(horizontal: 18.0, vertical: 30),
            child: Form(
              key: _formKey,
              child: Column(
                children: [
                              _buildTextField('Full Name', _nameController, false,
Validator.validateName, _nameFocusNode),
                  const SizedBox(height: 10),
                                  _buildTextField('Email', _emailController, false,
Validator.validateEmail, _emailFocusNode),
                  const SizedBox(height: 10),
                                _buildTextField('Password', _passwordController, true,
Validator.validatePassword, _passwordFocusNode),
                  const SizedBox(height: 10),
                    _buildTextField('Confirm Password', _confirmPasswordController,
true, (value) {
```

```
                                    return  Validator.validateConfirmPassword(value  ??  '',
_passwordController.text);
                  }, _confirmPasswordFocusNode),
                  const SizedBox(height: 50),
                  GestureDetector(
                    onTap: _isLoading ? null : _registerUser,
                    child: Container(
                      height: 55,
                      width: 300,
                      decoration: BoxDecoration(
                        borderRadius: BorderRadius.circular(30),
                        gradient: const LinearGradient(
                          colors: [AppColors.primaryColor, AppColors.secondaryColor],
                        ),
                      ),
                      child: Center(
                        child: _isLoading
                            ? const CircularProgressIndicator(color: Colors.white)
                            : const Text(
                          'SIGN UP',
                                style: TextStyle(fontWeight: FontWeight.bold, fontSize: 20,
color: AppColors.buttonTextColor),
                        ),
                      ),
                    ),
                  ),
                  const SizedBox(height: 50),
                ],
              ),
            ),
          ),
        ),
      ],
    ),
```

```
    );
  }
```

Output:


Hello
Sign in!
Gmail
Password
Forgot Password?
SIGN IN
Don't have an account?
Sign up



Conclusion: We learned how to include icons, images, and custom fonts in a Flutter application to improve the visual quality and user experience.

● Icons help to visually communicate actions or features to users, enhancing the user interface.

● Images are a powerful tool for adding visual context, branding, and dynamic content.

● Custom fonts allow you to personalize the app's text style and ensure it aligns with your brand identity.

Mastering the integration of these elements is essential for developing visually appealing and user-friendly Flutter applications.

MAD & PWA Lab Journal

Experiment No.	04
Experiment Title.	To create an interactive Form using form widget
Roll No.	46
Name	SAACHI RAHEJA
Class	D15B
Subject	MAD & PWA Lab
Lab Outcome	LO2: Design and Develop interactive Flutter App by using widgets, layouts, gestures and animation
Grade:	

Aim:To create an interactive Form using form widget

Theory: In Flutter, forms are created using the Form widget, which serves as a container for input fields (like TextFormField) and allows you to handle the validation and submission of data.
Key Components:
1. Form Widget:
○ The Form widget is used to group multiple input fields. It provides the ability to manage and validate the form data. ○ A GlobalKey<FormState> is required to track the form's state for validation and saving the data.
2. TextFormField:
○ TextFormField is the primary widget for collecting text input in Flutter forms.
○ It comes with built-in support for validation and user input handling.
3. Validation:
○ The validator property of the TextFormField allows you to define rules that ensure user inputs are valid (e.g., email format, required fields).
○ Flutter will automatically show error messages when the validation fails.
4. FormState:
○ To manage the state of the form, you use FormState which allows you to validate, save, and reset the form.
○ You can trigger validation by calling formKey.currentState?.validate().
5. Saving Data:
○ Once the form is valid, you can save the data using the onSaved property of the TextFormField.
Steps to Create an Interactive Form:
1. Create a Form Widget:
○ You use the Form widget to wrap the form fields and manage validation.
○ A GlobalKey<FormState> is used to access the form's state. 2. Add Form Fields (TextFormField):
○ For each input field, you use the TextFormField widget. ○ Each TextFormField can have a validator to ensure the input is valid (e.g., ensuring that the user provides a valid email, password, etc.).
○ Use onSaved to store the user input when the form is submitted.
3. Validate the Form:

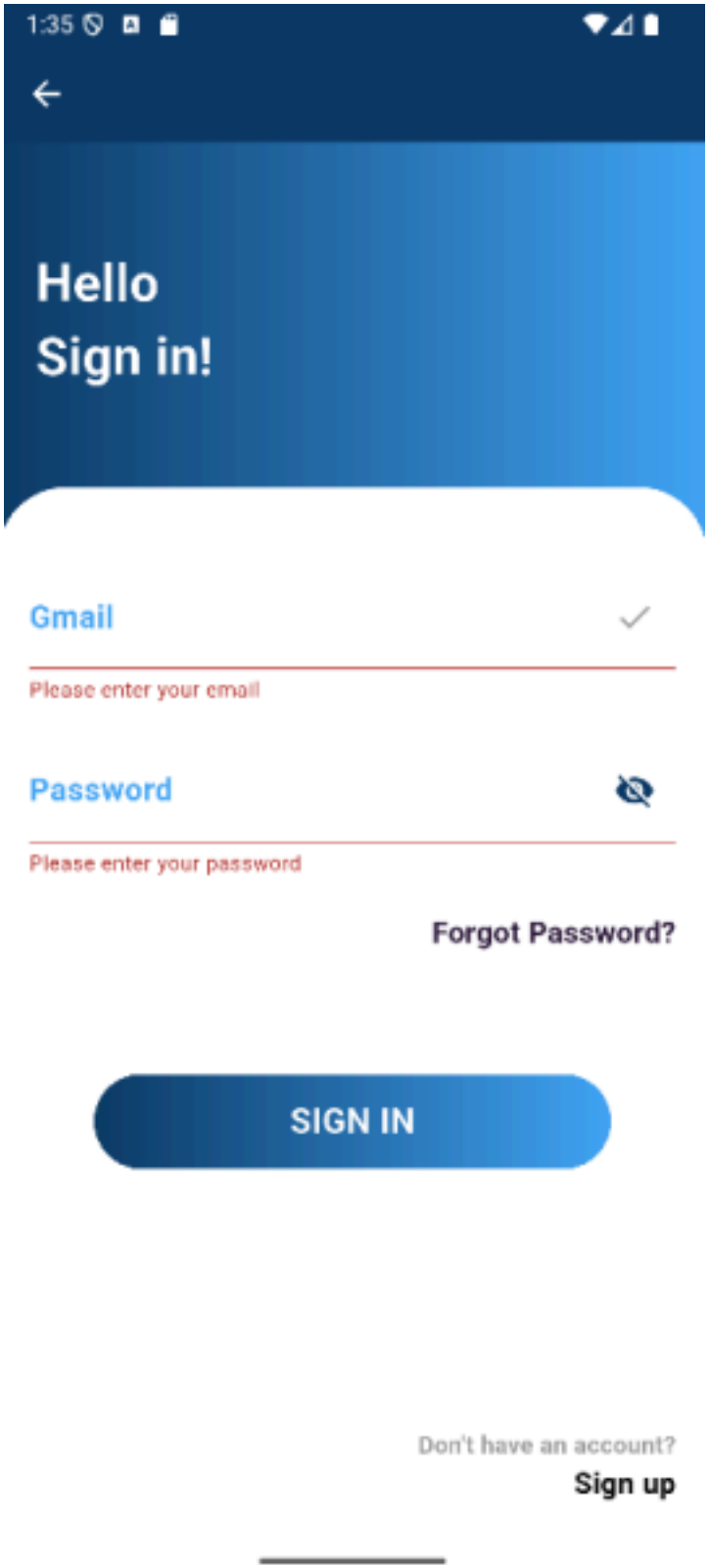
○ You can validate the form using the formKey.currentState?.validate() method, which triggers the validator for each field. If any field is invalid, it prevents form submission.
4. Handle Form Submission:
○ Once the form is validated, the form data is saved by calling formKey.currentState?.save().
○ The form can then be processed (e.g., sending data to a server, storing locally).

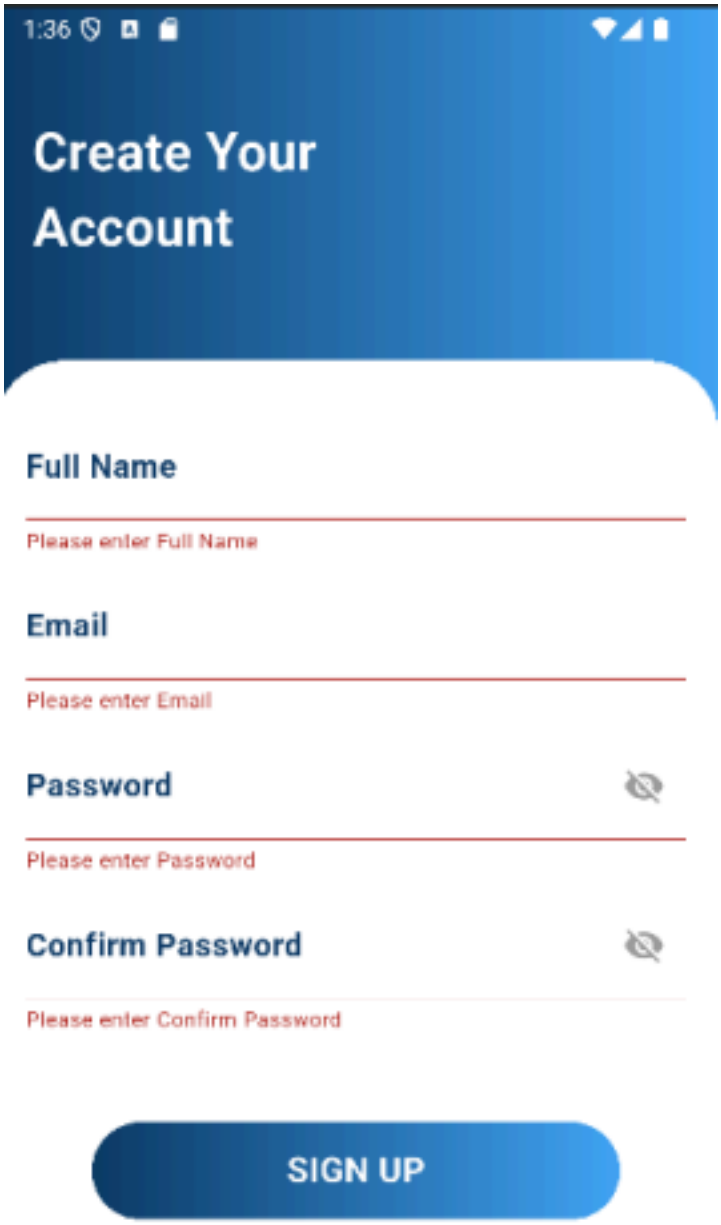
Code:

```
class Validator {
  static String? validateEmail(String? value) {
    if (value == null || value.isEmpty) {
      return 'Please enter your email';
    }
    // Regex for validating email format
    final emailRegex =
RegExp(r"^[a-zA-Z0-9._%+-]+@[a-zA-Z0-9.-]+\.[a-zA-Z]{2,}$");
    if (!emailRegex.hasMatch(value)) {
      return 'Please enter a valid email address';
    }
    return null;
  }

  static String? validatePassword(String? value) {
    if (value == null || value.isEmpty) {
      return 'Please enter your password';
    }

    if (value.length < 6) {
      return 'Password must be at least 6 characters long';
    }

    final regex = RegExp(r'^(?=.*[A-Z])(?=.*[a-z])(?=.*\d).{6,}$');
    if (!regex.hasMatch(value)) {
```

```
      return 'Password must contain at least one uppercase letter, one lowercase letter, and one number';
    }

    return null;
  }


  static String? validateName(String? value) {
    if(value == null || value.isEmpty) {
      return 'Please enter a valid name';
    }
    if(value.length < 2){
      return 'Name must be at least 2 characters long';
    }
    return null;
  }

  static String? validateConfirmPassword(String value, String password) {
    if (value.isEmpty) {
      return 'Confirm password cannot be empty';
    }
    if (value != password) {
      return 'Passwords do not match';
    }
    return null;
  }
}
```

Output:


1:35
Hello
Sign in!
Gmail
Please enter your email
Password
Please enter your password
Forgot Password?
SIGN IN
Don't have an account?
Sign up


1:36
Create Your Account
Full Name
Please enter Full Name
Email
Please enter Email
Password
Please enter Password
Confirm Password
Please enter Confirm Password
SIGN UP

Conclusion

In this experiment, you learned how to create an interactive form in Flutter using the Form widget. The Form widget allows for easy management of form fields, validation, and submission, while TextFormField provides the necessary functionality for input fields. By using validation and form state management, you can ensure that data entered by the user is valid before proceeding with any further operations, such as sending the data to a backend or storing it locally. This is a fundamental concept in Flutter for collecting and processing user input effectively.

MAD & PWA Lab Journal

Experiment No.	05
Experiment Title.	To apply navigation, routing and gestures in Flutter App
Roll No.	46
Name	SAACHI RAHEJA
Class	D15B
Subject	MAD & PWA Lab
Lab Outcome	LO2: Design and Develop interactive Flutter App by using widgets, layouts, gestures and animation
Grade:	

Aim: To apply navigation, routing and gestures in Flutter App
Theory :
1. Navigation in Flutter
Navigation refers to the mechanism that allows users to move between different screens or pages within a Flutter app. Flutter uses the Navigator widget to manage a stack of screens, providing a way to push and pop screens.
Key Concepts:
● Navigator: The core widget for managing a stack of routes.
● Routes: Represent different screens in an app.
Push and Pop Navigation:
Push: Navigates to a new screen.

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

Pop: Returns to the previous screen.

```
Navigator.pop(context);
```

Named Routes:
Using named routes improves code readability:

```
Navigator.pushNamed(context, '/second');
```

2. Routing in Flutter
Routing refers to how screens or pages are mapped and displayed in an app. In Flutter, routing is managed by the MaterialApp or CupertinoApp widget, which uses routes to determine which screen to show.
Types of Routes:
Static Routes: Defined at app launch.

```
MaterialApp(
routes: {
'/': (context) => HomeScreen(),
'/second': (context) => SecondScreen(),
```

},
);
Dynamic Routes: Allow passing parameters when navigating
Navigator.push(
context,
MaterialPageRoute(builder: (context) => DetailScreen(itemId: 42)),
);
3. Gestures in Flutter
Gestures are actions performed by the user on the screen, like tapping, swiping, or pinching. Flutter provides the GestureDetector widget to detect these actions.
Common Gestures:
Tap Gesture: Detects taps.
GestureDetector(onTap: () => print("Tapped"), child: Container());
Long Press Gesture: Detects long presses.
GestureDetector(onLongPress: () => print("Long Pressed"), child: Container());
Swipe Gesture: Detects swipes (horizontal or vertical).
GestureDetector(
onHorizontalDragEnd: (details) => print("Swiped"), child: Container(),
);
Combining Navigation and Gestures:
Gestures can also trigger navigation between screens.
GestureDetector(
onHorizontalDragEnd: (details) =>
Navigator.pushNamed(context, '/nextScreen'), child: Container(),
);
Code:
import 'package:flutter/material.dart';

```
import 'package:firebase_auth/firebase_auth.dart';
import 'package:fluttertoast/fluttertoast.dart'; // Import
FlutterToast
import 'login_screen.dart'; // Import Login screen
import '../app_colors.dart'; // Import AppColors

class VerifyEmailScreen extends StatelessWidget {
  VerifyEmailScreen({Key? key}) : super(key: key);

  Future<void> _verifyEmail(BuildContext context) async {
    try {
      User? user = FirebaseAuth.instance.currentUser;

      if (user != null) {
        // Check if the email is verified
        await user.reload();
        user = FirebaseAuth.instance.currentUser;

        if (user?.emailVerified == true) {
          // Show success toast message for email verification
          Fluttertoast.showToast(
            msg: "Email Verified Successfully!",
            toastLength: Toast.LENGTH_SHORT,
            gravity: ToastGravity.BOTTOM,
            timeInSecForIosWeb: 1,
            backgroundColor: Colors.green,
            textColor: Colors.white,
            fontSize: 16.0,
          );

          // Navigate to the login screen
          Navigator.pushReplacement(
```

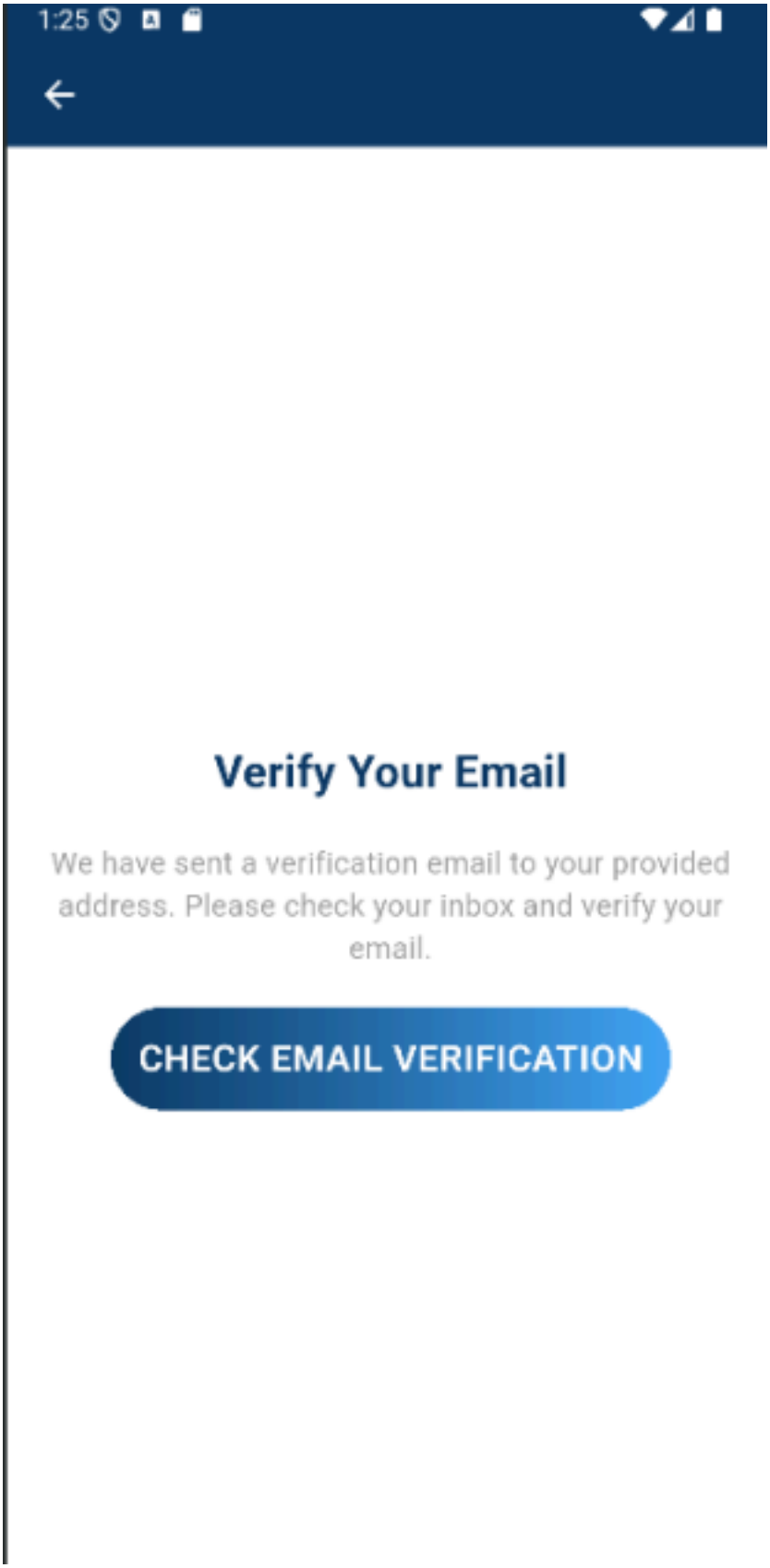
```
          context,
          MaterialPageRoute(
            builder: (context) => const LoginScreen(),
          ),
        );
      } else {
        // Show a toast if email is not verified
        Fluttertoast.showToast(
          msg: "Email is not verified. Please check your inbox.",
          toastLength: Toast.LENGTH_SHORT,
          gravity: ToastGravity.BOTTOM,
          timeInSecForIosWeb: 1,
          backgroundColor: Colors.orange,
          textColor: Colors.white,
          fontSize: 16.0,
        );
      }
    } else {
      // Show toast if no user found
      Fluttertoast.showToast(
        msg: "No user found or email already verified!",
        toastLength: Toast.LENGTH_SHORT,
        gravity: ToastGravity.BOTTOM,
        timeInSecForIosWeb: 1,
        backgroundColor: Colors.red,
        textColor: Colors.white,
        fontSize: 16.0,
      );
    }
  } catch (e) {
    // Show error toast in case of exception
    Fluttertoast.showToast(
```

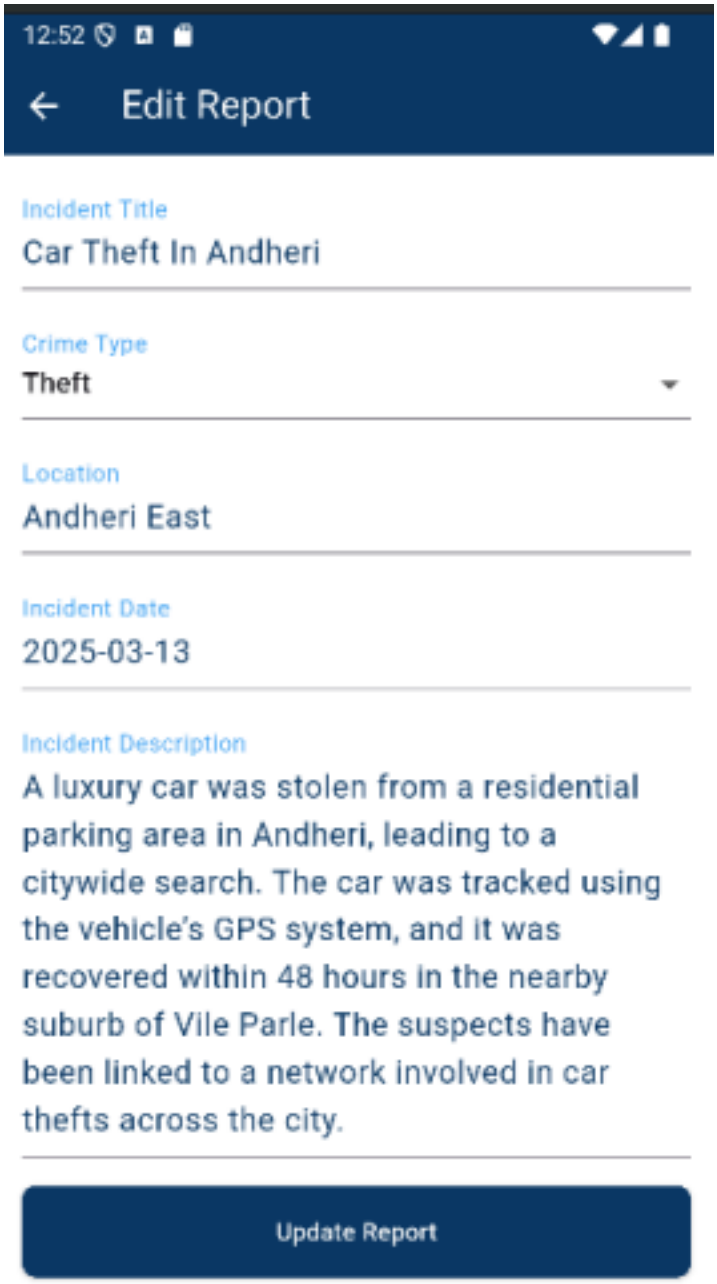
```
      msg: "Error: ${e.toString()}",
      toastLength: Toast.LENGTH_SHORT,
      gravity: ToastGravity.BOTTOM,
      timeInSecForIosWeb: 1,
      backgroundColor: Colors.red,
      textColor: Colors.white,
      fontSize: 16.0,
    );
  }
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      leading: IconButton(
        icon: const Icon(Icons.arrow_back),
        onPressed: () {
          Navigator.pop(context);
        },
        color: AppColors.backgroundColor,
      ),
      backgroundColor: AppColors.primaryColor,
      elevation: 0,
    ),
    body: Padding(
      padding: const EdgeInsets.all(20.0),
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          const Text(
            'Verify Your Email',
```

```
            style: TextStyle(
              fontSize: 24,
              fontWeight: FontWeight.bold,
              color: AppColors.primaryColor,
            ),
          ),
          const SizedBox(height: 20),
          const Text(
            'We have sent a verification email to your provided
address. Please check your inbox and verify your email.',
            textAlign: TextAlign.center,
            style: TextStyle(
              fontSize: 16,
              color: Colors.grey,
            ),
          ),
          const SizedBox(height: 20),
          GestureDetector(
            onTap: () {
              _verifyEmail(context);
            },
            child: Container(
              height: 55,
              width: 300,
              decoration: BoxDecoration(
                borderRadius: BorderRadius.circular(30),
                gradient: const LinearGradient(
                  colors: [AppColors.primaryColor,
AppColors.secondaryColor],
                ),
              ),
              child: const Center(
```

```
                child: Text(
                  'CHECK EMAIL VERIFICATION',
                  style: TextStyle(
                    fontWeight: FontWeight.bold,
                    fontSize: 20,
                    color: AppColors.backgroundColor,
                  ),
                ),
              ),
            ),
          ),
        ],
      ),
    ),
  );
 }
}
```

Output:


1:25
Verify Your Email
We have sent a verification email to your provided address. Please check your inbox and verify your email.
CHECK EMAIL VERIFICATION


12:52
Edit Report
Incident Title
Car Theft In Andheri
Crime Type
Theft
Location
Andheri East
Incident Date
2025-03-13
Incident Description
A luxury car was stolen from a residential parking area in Andheri, leading to a citywide search. The car was tracked using the vehicle's GPS system, and it was recovered within 48 hours in the nearby suburb of Vile Parle. The suspects have been linked to a network involved in car thefts across the city.
Update Report

Conclusion

In Flutter, navigation, routing, and gestures are integral for creating interactive and user-friendly applications.

- Navigation enables seamless transitions between screens, using methods like push and pop.
- Routing allows for both static and dynamic screen management, enhancing flexibility and modularity in app design.
- Gestures offer a way for users to interact with the app, making the experience more engaging.

By mastering these concepts, developers can build well-structured, responsive, and intuitive applications in Flutter. These techniques are essential for handling user interactions, creating dynamic navigation flows, and optimizing the overall user experience in any mobile app.

MAD & PWA Lab Journal

Experiment No.	06
Experiment Title.	To Connect Flutter UI with fireBase database
Roll No.	46
Name	SAACHI RAHEJA
Class	D15B
Subject	MAD & PWA Lab
Lab Outcome	LO3: Analyze and Build production ready Flutter App by incorporating backend services and deploying on Android / iOS
Grade:	

Aim: To connect a Flutter application to Firebase by integrating Firebase Core, and initializing Firebase services in a Flutter project.

Theory:Firebase provides several backend services like Authentication, Firestore, Realtime Database, Storage, etc., to build mobile and web apps. Firebase Core is the essential service that allows Flutter apps to communicate with Firebase. Once Firebase Core is connected, you can add specific Firebase services like Firestore, Firebase Authentication, Firebase Storage, etc.
This experiment focuses on:
1. Setting up Firebase for your project.
2. Integrating Firebase Core with a Flutter app.
3. Platform-specific configurations (Android and iOS). 4. Basic setup for Firebase services.
Steps to Connect Firebase with Flutter:
1. Firebase Console Setup:
To begin with, you'll need a Firebase project.
Create Firebase Project:
1. Go to Firebase Console.
2. Click on Create a New Project and follow the steps.
3. After creating the project, you'll be directed to the Firebase console for your project.
4. Enable the Firebase services you need (e.g., Firestore, Firebase Authentication, Firebase Storage).
2. Add Your Flutter App to Firebase:
For Android:
1. Register Your Android App:
○ In the Firebase Console, click Add app and choose Android.
○ Register your app with your Android package name (you can find it in android/app/src/main/AndroidManifest.xml).
○ Download the google-services.json file.
2. Place google-services.json in the Android directory:
○ Put the downloaded google-services.json file in the android/app/ directory.
3. Configure Android build files:
In android/build.gradle, add the following classpath inside the dependencies block:

classpath 'com.google.gms:google-services:4.3.3' // Add this line
Then, in the android/app/build.gradle file, add the following line at the bottom of the file:
apply plugin: 'com.google.gms.google-services' // Add this line
For iOS:
1. Register Your iOS App:
○ In the Firebase Console, click Add app and choose iOS.
○ Register the iOS app with your iOS bundle ID.
○ Download the GoogleService-Info.plist file.
2. Add GoogleService-Info.plist to Xcode:
○ Open your Flutter project in Xcode.
○ Drag the GoogleService-Info.plist into the Runner project inside Xcode.
○ Make sure to select Copy items if needed and add it to your app target.
3. Add Firebase SDK in iOS:
In the ios/Podfile, ensure you have the following lines:
platform :ios, '10.0' # Firebase SDK requires at least iOS 10
Run the following command to install the CocoaPods dependencies:
cd ios
pod install
cd ..
3. Add Firebase Dependencies in Flutter:
In your pubspec.yaml file, include the following dependencies to initialize Firebase Core:
dependencies:
flutter:
sdk: flutter
firebase_core: ^1.10.0 # Firebase Core
Run the following command to install the dependencies:
flutter pub get
4. Initialize Firebase in Flutter:
In the main.dart file, you'll need to initialize Firebase before you use any Firebase service. Add the following code:
import 'package:flutter/material.dart';
import 'package:firebase_core/firebase_core.dart'; // Import Firebase Core
void main() async {

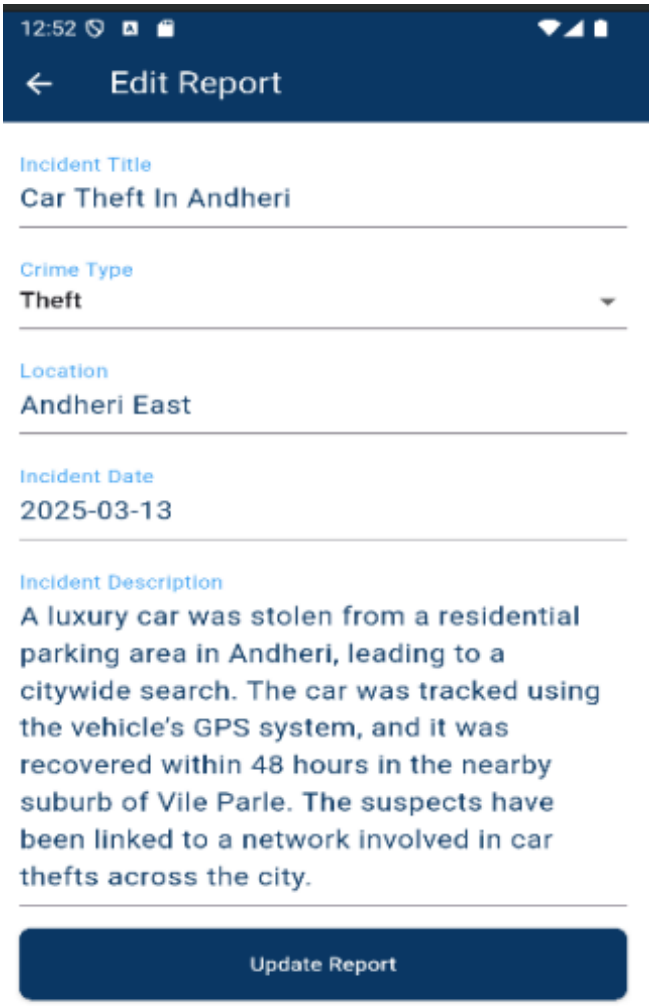
```
WidgetsFlutterBinding.ensureInitialized(); // Ensures Firebase is initialized before
app starts await Firebase.initializeApp(); // Initialize Firebase
runApp(MyApp());
}
class MyApp extends StatelessWidget {
@override
Widget build(BuildContext context) {
return MaterialApp(
title: 'Flutter Firebase Connection',
theme: ThemeData(
primarySwatch: Colors.blue,
),
home: Scaffold(
appBar: AppBar(
title: Text('Firebase Initialized'),
),
body: Center(
child: Text('Firebase Connected
Successfully!'),
),
),
);
}
}
```

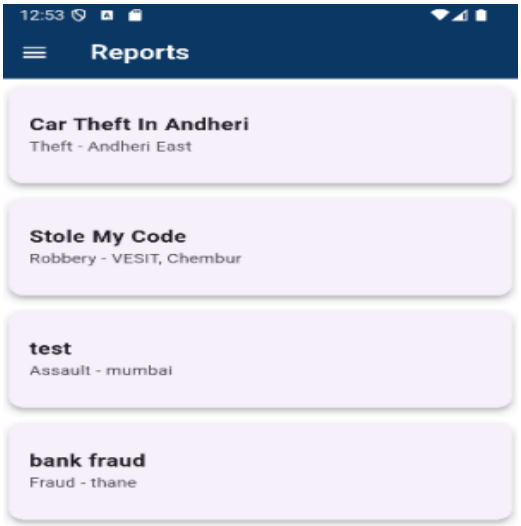
5. Handle Platform-Specific Setup (for Android and iOS): For Android: Ensure the following is in your android/app/src/main/AndroidManifest.xml to allow Firebase services:

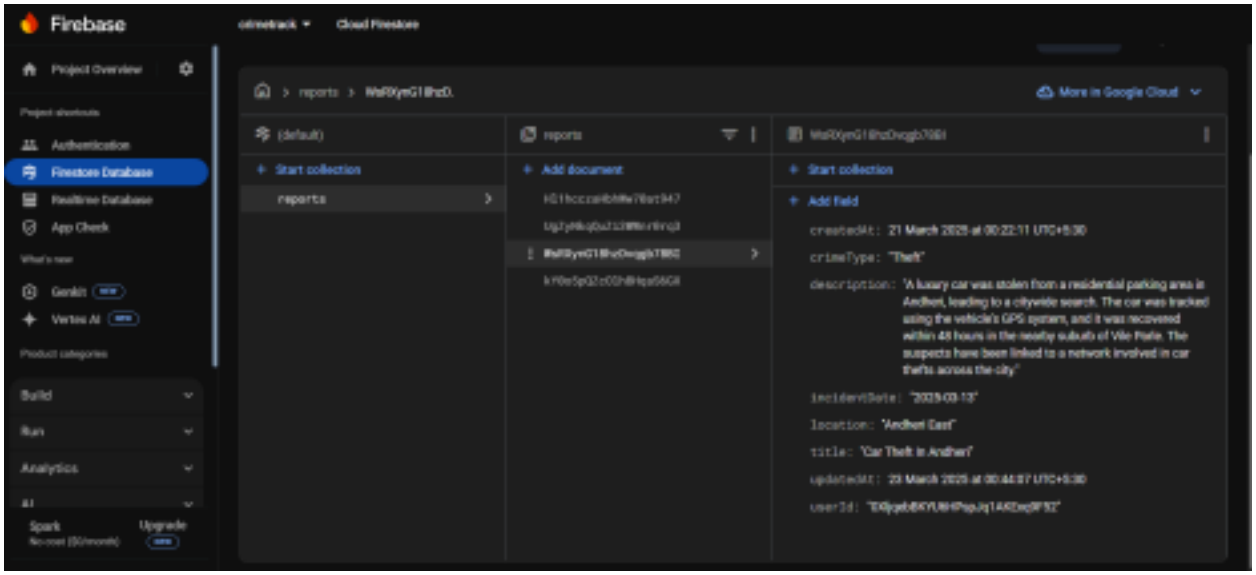
```
<application
android:label="your-app-name"
android:icon="@mipmap/ic_launcher">
<!-- Add this line for Firebase services --> <meta-data
android:name="com.google.firebase.messaging.default_not ification_icon"
android:resource="@drawable/ic_notification" /> </application>
```

For iOS:

You may need to request permissions for notifications or other Firebase services in your Info.plist file (if using Firebase Messaging, for example).
<key>UIBackgroundModes</key>
<array>
<string>fetch</string>
<string>remote-notification</string>
</array>
<key>NSLocationWhenInUseUsageDescription</key> <string>Your app requires access to location</string>
6. Testing Firebase Connection:
Run the app on an emulator or physical device. If everything is set up correctly, the app should launch with the message: "Firebase Connected Successfully!"
7. Optional: Firebase Services Integration (Firestore, Firebase Auth, etc.)
After successfully connecting Firebase, you can easily integrate other Firebase services, such as:
1. Firebase Firestore:
○ Add the cloud_firestore package in pubspec.yaml and perform operations (CRUD).
2. Firebase Authentication:
○ Add the firebase_auth package and enable authentication methods like email/password or Google sign-in.
3. Firebase Cloud Storage:
○ Use the firebase_storage package for storing and retrieving files.
Output:


12:52
Edit Report
Incident Title
Car Theft In Andheri
Crime Type
Theft
Location
Andheri East
Incident Date
2025-03-13
Incident Description
A luxury car was stolen from a residential parking area in Andheri, leading to a citywide search. The car was tracked using the vehicle's GPS system, and it was recovered within 48 hours in the nearby suburb of Vile Parle. The suspects have been linked to a network involved in car thefts across the city.
Update Report





Conclusion:

After completing the above steps, your Flutter app is successfully connected to Firebase. The Firebase Core initialization allows your Flutter app to communicate with Firebase services. This setup can be expanded to include services like Firestore, Firebase Auth, Cloud Storage, and more based on the app's requirements.

MAD & PWA Lab Journal

Experiment No.	07
Experiment Title.	To write meta data of your Ecommerce PWA in a Web app manifest file to enable “add to homescreen feature”.
Roll No.	46
Name	SAACHI RAHEJA
Class	D15B
Subject	MAD & PWA Lab
Lab Outcome	LO4: Understand various PWA frameworks and their requirements
Grade:	

Aim :To write metadata for an eCommerce Progressive Web App (PWA) in a web app manifest file to enable the “Add to Home Screen” feature.

Theory: A Web App Manifest is a JSON file that contains essential metadata about a Progressive Web App (PWA). This metadata allows browsers to recognize and display the web app appropriately when a user chooses to install it. For the "Add to Home Screen" feature to function, the manifest file must define key details, such as the app name, icons, and display preferences.

Key Components of a Web App Manifest:

1. name: The full name of the app, displayed when users add the app to their home screen. It should be clear, descriptive, and concise.
2. short_name: A shorter version of the app’s name, used when there isn't enough space (e.g., on the home screen or app launcher).
3. description: A brief explanation of the app's purpose. It gives users an idea of what to expect when they add the app to their home screen.
4. start_url: Defines the entry point for the app. When the app is launched from the home screen, this URL is opened first.
5. display: Specifies how the app should appear when launched. Options include:
    - standalone: App behaves like a native app with no browser chrome.
    - fullscreen: App runs in full-screen mode.
    - minimal-ui: App provides a minimal UI with navigation controls.
    - browser: App behaves like a regular web page in a browser.
6. background_color: The background color that appears during the app's splash screen as it loads.
7. theme_color: Sets the color of the browser's UI elements, such as the address bar, to match the app's branding.
8. orientation: Specifies whether the app should open in portrait or landscape mode, improving the experience on mobile devices.
9. scope: Defines the navigation scope for the PWA. It limits the URLs that are considered within the app's domain.
10. icons: Specifies the various icon images used for the app on the home screen, app launcher, or other areas. Icons should be provided in multiple sizes for different devices and resolutions.

Importance for PWAs:

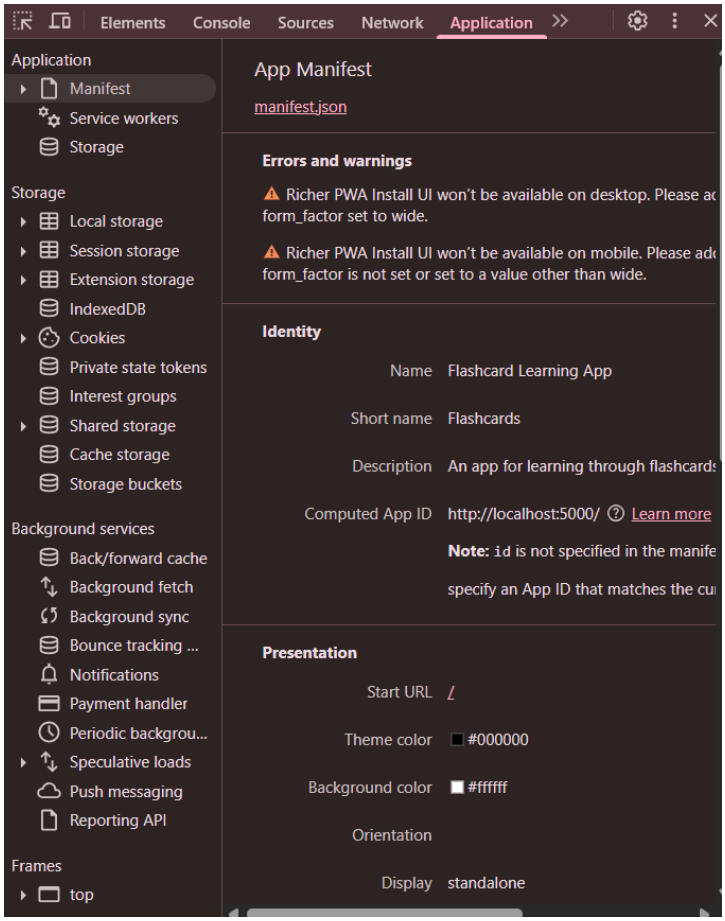
The web app manifest is fundamental for delivering a native-like experience on mobile and desktop devices. It gives users a smooth, app-like feel while ensuring that the app is visually consistent across different platforms. For eCommerce PWAs, this is crucial to enhance user experience, especially in terms of engagement and accessibility.

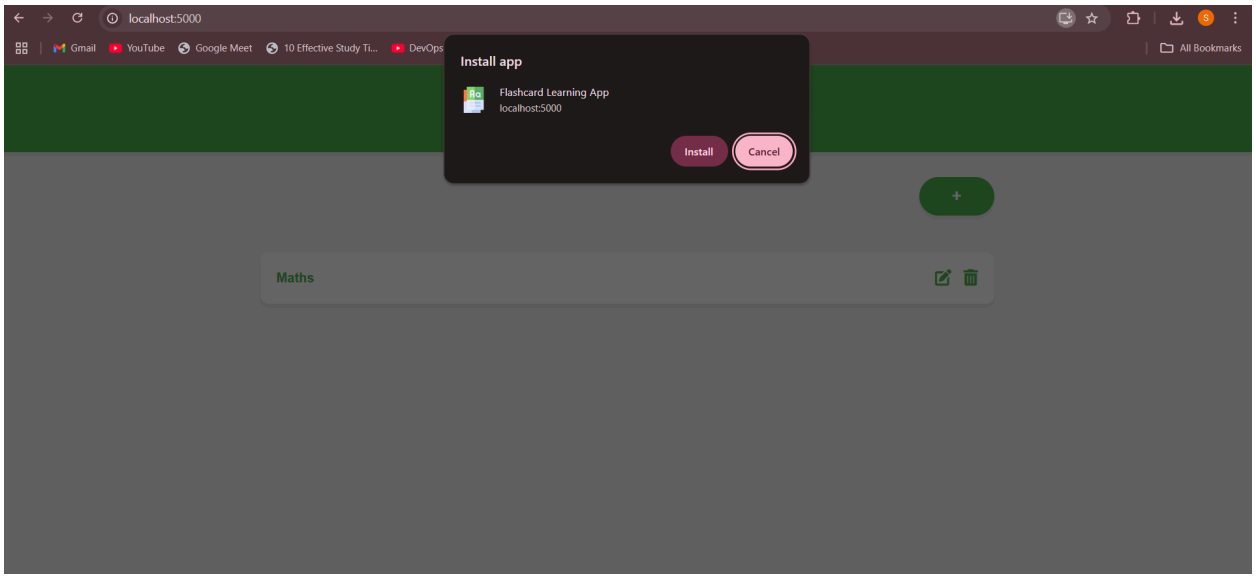
Codes:

Manifest.json

```
{
  "name": "Flashcard Learning App",
  "short_name": "Flashcards",
  "description": "An app for learning through flashcards with Quiz and Study modes",
  "start_url": "/",
  "display": "standalone",
  "background_color": "#ffffff",
  "theme_color": "#000000",
  "icons": [
    {
      "src": "/static/images/icon-128x128.png",
      "sizes": "128x128",
      "type": "image/png"
    },
    {
      "src": "/static/images/icon-512x512.png",
      "sizes": "512x512",
      "type": "image/png"
    }
  ]
}
```

Screenshots:





Conclusion: Writing the metadata in the Web App Manifest file is vital for enabling the “Add to Home Screen” feature in an eCommerce PWA. By ensuring that essential details like the app's name, icons, start URL, and display settings are correctly defined, users can enjoy a seamless experience when adding the app to their home screen. This integration is key for increasing user retention and engagement, as it allows eCommerce platforms to deliver an immersive, fast, and convenient browsing experience.

MAD & PWA Lab Journal

Experiment No.	08
Experiment Title.	To code and register a service worker, and complete the install and activation process for a new service worker for the E-commerce PWA
Roll No.	46
Name	SAACHI RAHEJA
Class	D15B
Subject	MAD & PWA Lab
Lab Outcome	LO5: Design and Develop a responsive User Interface by applying PWA Design techniques
Grade:	

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 script that runs in the background of the web browser, independent of the web page. It acts as a network proxy, allowing the app to manage caching, intercept network requests, and enable offline functionality. Service workers are essential for PWAs to function offline and to provide fast loading times.

The process of implementing a service worker for an E-commerce PWA includes:

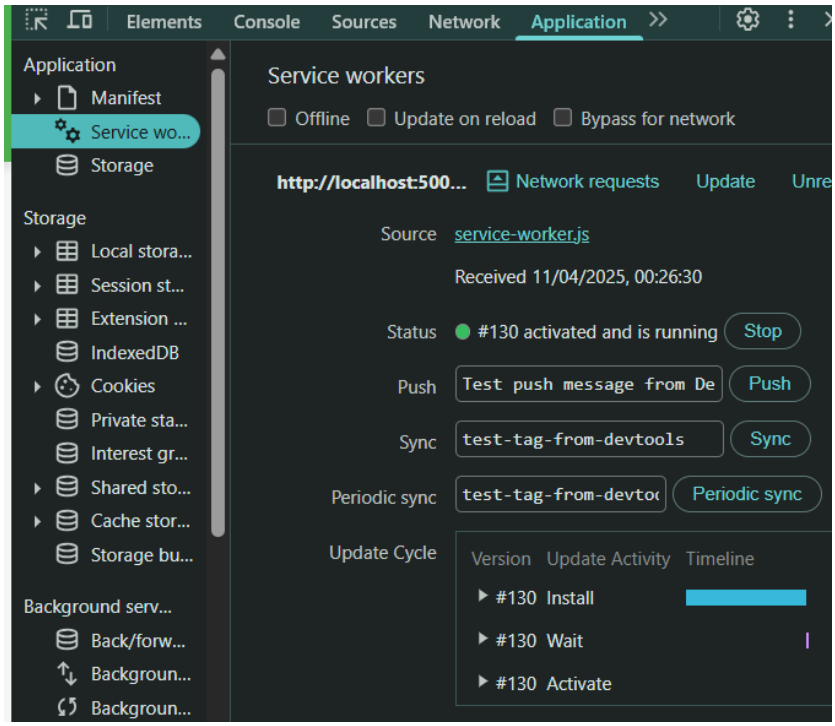
1. Coding the Service Worker: The service worker script typically handles tasks like caching static assets, intercepting network requests, and managing updates.
2. Registering the Service Worker: To enable the service worker, it must be registered in the main JavaScript file of the application.
3. Install Event: The install event occurs when the service worker is first installed. During this event, caching of essential assets like HTML, CSS, JS, and images happens.
4. Activate Event: After installation, the service worker goes through the activation phase. During activation, the previous service workers (if any) are deleted, and the new worker becomes active.
5. Update Process: The service worker can be updated if the script changes. When an update is available, the old service worker is terminated, and the new one is activated.

Code:

```
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open('v1').then(function(cache) {
      return cache.addAll([
        '/',
        '/static/manifest.json',
        '/static/js/script.js',
         '/static/images/icon-128x128.png',
         '/static/images/icon-512x512.png',
      ]);
    })
```

```
  );
});

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

Screenshots:



Conclusion

Implementing and registering a service worker for the E-commerce PWA is crucial for enabling offline capabilities, improving performance, and managing caching. Through the install and activation process, the PWA can provide a seamless experience for users even without a network connection.

MAD & PWA Lab Journal

Experiment No.	09
Experiment Title.	To implement Service worker events like fetch, sync and push for E-commerce PWA
Roll No.	46
Name	SAACHI RAHEJA
Class	D15B
Subject	MAD & PWA Lab
Lab Outcome	LO5: Design and Develop a responsive User Interface by applying PWA Design techniques
Grade:	

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

Theory:Service workers provide powerful features that allow web applications to operate reliably, even under unreliable network conditions. Key events such as fetch, sync, and push enhance the performance, offline capability, and user engagement of a Progressive Web App (PWA).

1. Fetch Event

The fetch event allows the service worker to intercept network requests made by the PWA. This is commonly used for caching strategies, such as:

- Cache-first: Serve content from cache, then update in the background.
- Network-first: Try to fetch from the network, fall back to cache if offline.

This is useful for delivering product pages, images, or static assets quickly and reliably.

2. Sync Event

The sync event, especially background sync, helps the app manage tasks when connectivity is restored. For instance, if a user submits an order or review while offline, the service worker can save it locally and send it once the connection is re-established.

- Requires registering a sync task using registration.sync.register('tag-name').

3. Push Event

The push event enables the PWA to receive push notifications from a server, even when the app is not open. This is ideal for eCommerce apps to send:

- Order updates
- Promotional offers
- Cart reminders

It requires integration with a push service and permission from the user.

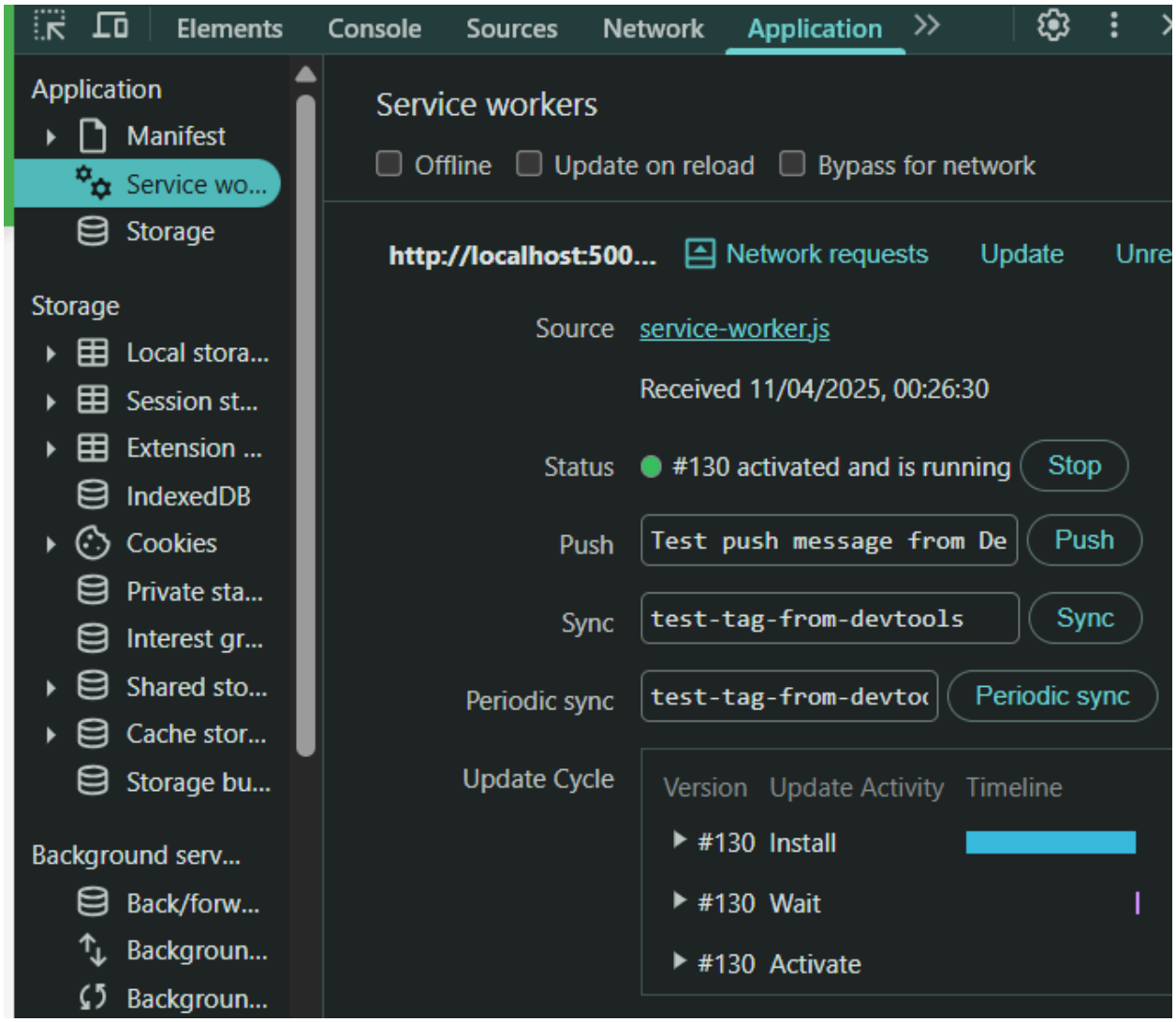
Code:

```
self.addEventListener('install', function(event) {
  event.waitUntil(
    caches.open('flashcard-cache-v1').then(function(cache) {
      return cache.addAll([
        '/',
        '/static/manifest.json',
        '/static/js/script.js',
        '/static/images/icon-128x128.png',
        '/static/images/icon-512x512.png'
      ]);
    })
  );
});

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

self.addEventListener('activate', function(event) {
  const cacheWhitelist = ['flashcard-cache-v1'];
  event.waitUntil(
    caches.keys().then(function(cacheNames) {
      return Promise.all(
        cacheNames.map(function(cacheName) {
          if (!cacheWhitelist.includes(cacheName)) {
            return caches.delete(cacheName);
          }
        })
      );
    })
```

```
  );
});

// Background Sync - For syncing offline-created flashcards or decks
self.addEventListener('sync', function(event) {
  if (event.tag === 'sync-flashcards') {
    event.waitUntil(syncFlashcardsToServer());
  }
});

async function syncFlashcardsToServer() {
  const cards = await getUnsyncedFlashcards(); // Replace with IndexedDB logic
  for (const card of cards) {
    try {
      const res = await fetch('/api/cards', {
        method: 'POST',
        body: JSON.stringify(card),
        headers: {
          'Content-Type': 'application/json'
        }
      });
      if (res.ok) {
        await markCardAsSynced(card.id);
      }
    } catch (err) {
      console.error('Sync failed for card', card.id, err);
    }
  }
}

// Push Notifications - For reminders or study tips
self.addEventListener('push', function(event) {
  const data = event.data ? event.data.json() : {};
  const options = {
    body: data.body || 'Time to study your flashcards!',
```

```
    icon: '/static/images/icon-128x128.png',
    badge: '/static/images/icon-128x128.png',
    data: {
      url: data.url || '/'
    }
  };

  event.waitUntil(
    self.registration.showNotification(data.title || 'Flashcard Reminder', options)
  );
});

self.addEventListener('notificationclick', function(event) {
  event.notification.close();
  event.waitUntil(
    clients.openWindow(event.notification.data.url)
  );
});

// Demo IndexedDB stubs — replace with real logic
async function getUnsyncedFlashcards() {
  return [{ id: 1, question: 'What is PWA?', answer: 'Progressive Web App' }];
}

async function markCardAsSynced(id) {
  console.log(`✅ Flashcard ${id} marked as synced.`);
}
```

Screenshots:



Conclusion

Implementing fetch, sync, and push events in the service worker of an E-commerce PWA significantly enhances the app's usability, resilience, and user engagement. These features enable the PWA to function offline, perform background tasks, and communicate with users proactively, providing a seamless and efficient shopping experience.

MAD & PWA Lab Journal

Experiment No.	10
Experiment Title.	To study and implement deployment of Ecommerce PWA to GitHub Pages.
Roll No.	
Name	
Class	D15B
Subject	MAD & PWA Lab
Lab Outcome	LO5: Design and Develop a responsive User Interface by applying PWA Design techniques
Grade:	

MAD & PWA Lab Journal

Experiment No.	11
Experiment Title.	To use google Lighthouse PWA Analysis Tool to test the PWA functioning.
Roll No.	46
Name	SAACHI RAHEJA
Class	D15B
Subject	MAD & PWA Lab
Lab Outcome	LO6: Develop and Analyze PWA Features and deploy it over app hosting solution
Grade:	

Aim:To use the Google Lighthouse PWA Analysis Tool to test the Progressive Web App (PWA) functionality.

Theory:Progressive Web Apps (PWAs) are web applications designed to provide a seamless, app-like experience on the web. Key characteristics of PWAs include:

1. Responsive Design: PWAs automatically adjust to different screen sizes and devices, providing a consistent user experience across mobile, tablet, and desktop.
2. Offline Capabilities: PWAs can function without an internet connection, thanks to service workers that cache assets and enable offline functionality.
3. App-like Experience: PWAs provide an app-like feel, including features such as home screen installation, push notifications, and improved loading performance.
4. Web App Manifest: A PWA includes a manifest file that defines how the app should appear when installed on the device, such as app name, icons, and theme color.

Google Lighthouse is an automated tool by Google that audits web pages for performance, accessibility, SEO, and best practices. Specifically, for PWAs, Lighthouse checks:

- Service Worker: Ensures that your app has a properly configured service worker for offline functionality.
- Web App Manifest: Validates that your app includes a manifest file with the necessary metadata for installation.
- Offline Functionality: Verifies that the app works offline by caching assets and supporting service worker operations.
- Performance Metrics: Assesses loading time, interactivity, and other critical aspects for a smooth user experience.

Objective:

- To evaluate the performance, PWA compliance, and best practices of the web app.
- To identify areas for improvement in the offline functionality, manifest file configuration, and service worker setup.

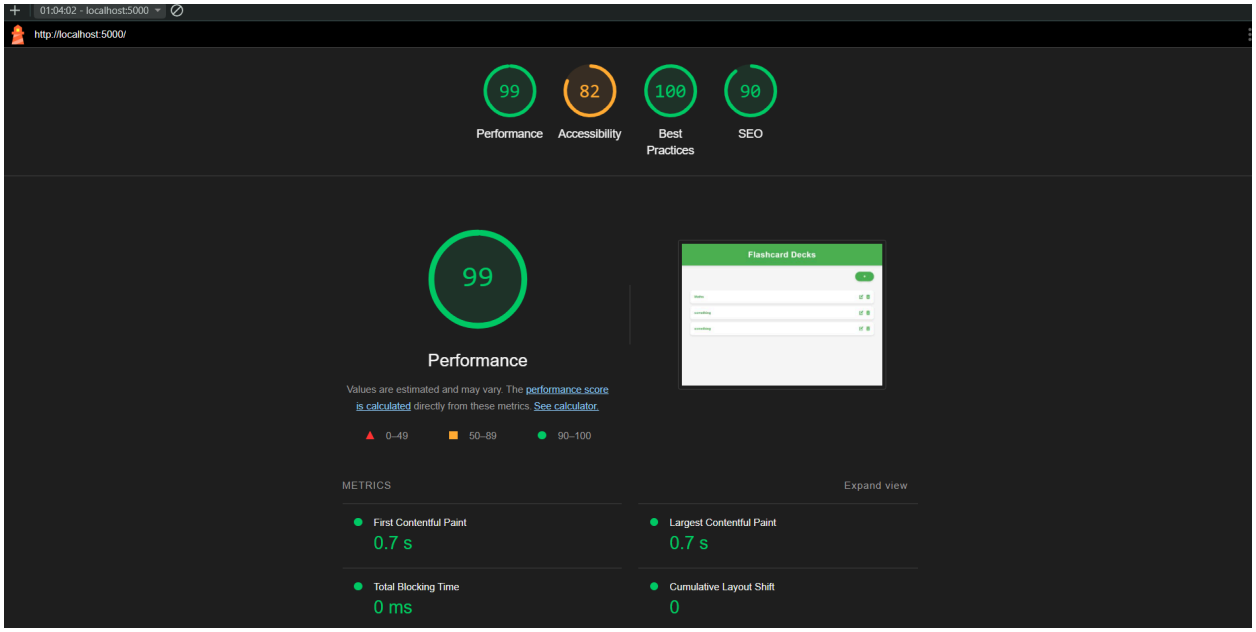
Steps to Use Google Lighthouse for PWA Analysis:

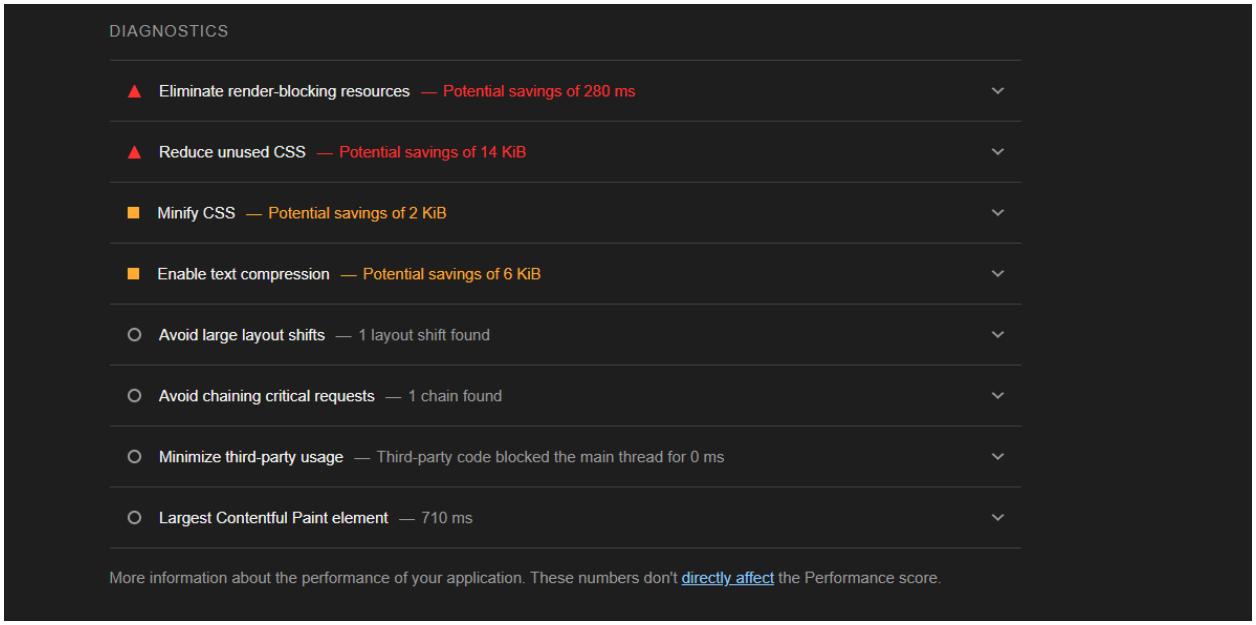
1. Open Google Chrome:
    - Make sure you're using Google Chrome, as Lighthouse is integrated into Chrome's DevTools.
2. Open Your PWA in Google Chrome:
    - Navigate to your PWA URL in Google Chrome.
3. Open Chrome DevTools:
    - Right-click anywhere on the page and select Inspect (or use the shortcut Ctrl+Shift+I on Windows/Linux or Cmd+Opt+I on macOS).
    - Click on the Lighthouse tab in the DevTools panel. If it’s not visible, click the » icon to find it.
4. Configure Lighthouse Settings:
    - Categories: Choose the categories you want to audit, including Performance, PWA, Accessibility, Best Practices, and SEO.
    - Device Mode: Select between Mobile or Desktop to simulate the audit on different devices.
    - Make sure to select the PWA category for a specific audit on Progressive Web App functionality.
5. Run the Lighthouse Audit:
    - Click the "Generate report" button to begin the audit. Lighthouse will analyze your PWA and generate a report.
6. Review the Results:
    - Once the audit is completed, you’ll see a detailed report with scores in different categories. In the PWA section, Lighthouse will evaluate:
        - Service Worker: Checks if a service worker is installed and functioning correctly.
        - Web App Manifest: Verifies the presence and correctness of your manifest file.
        - Offline Capability: Assesses if your PWA can function offline.
        - App Installability: Tests whether the app can be installed on the home screen.
7. Analyze Issues and Recommendations:

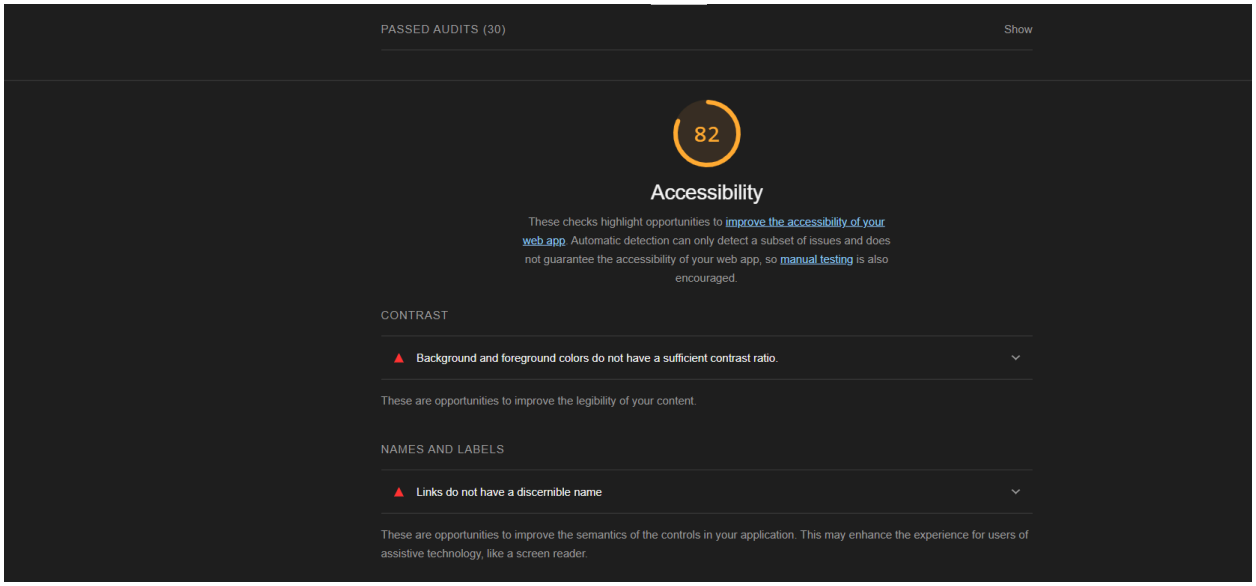
- Review the findings and follow Lighthouse's suggestions to improve areas such as offline support, manifest configuration, and service worker implementation.

8. Re-test After Fixes:
    - After applying fixes to the identified issues, re-run the Lighthouse audit to ensure the PWA now meets the required standards.

Screenshots:


01:04:02 - localhost:5000
http://localhost:5000/
99
Performance
82
Accessibility
100
Best Practices
90
SEO
99
Performance
Values are estimated and may vary. The performance score is calculated directly from these metrics. See calculator.
0–49
50–89
90–100
Flashcard Decks
METRICS
Expand view
First Contentful Paint
0.7 s
Largest Contentful Paint
0.7 s
Total Blocking Time
0 ms
Cumulative Layout Shift
0


DIAGNOSTICS
Eliminate render-blocking resources — Potential savings of 280 ms
Reduce unused CSS — Potential savings of 14 KiB
Minify CSS — Potential savings of 2 KiB
Enable text compression — Potential savings of 6 KiB
Avoid large layout shifts — 1 layout shift found
Avoid chaining critical requests — 1 chain found
Minimize third-party usage — Third-party code blocked the main thread for 0 ms
Largest Contentful Paint element — 710 ms
More information about the performance of your application. These numbers don't directly affect the Performance score.



Conclusion:

Using the Google Lighthouse PWA Analysis Tool is a crucial step in ensuring that your Progressive Web App meets key functionality standards. By following the steps above, you can identify and resolve any issues related to service workers, manifest files, and offline capabilities. This process helps optimize performance, enhances the user experience, and ensures compliance with PWA best practices.

MAD & PWA Lab Journal

Experiment No.	Assignment-1
Assignment 1 Questions	1. a) Explain the key features and advantages of using Flutter for mobile app development. b)Discuss how the Flutter framework differs from traditional approaches and why it has gained popularity in the developer community. 2. a)Describe the concept of the widget tree in Flutter. Explain how widget composition is used to build complex user interfaces. b) Provide examples of commonly used widgets and their roles in creating a widget tree. 3. a) Discuss the importance of state management in Flutter applications. b) Compare and contrast the different state management approaches available in Flutter, such as setState, Provider, and Riverpod. Provide scenarios where each approach is suitable. 4.a) Firebase Integration in Flutter: Explain the process of integrating Firebase with a Flutter application. b)Discuss the benefits of using Firebase as a backend solution. Highlight the Firebase services commonly used in Flutter development and provide a brief overview of how data synchronization is achieved.
Roll No.	46
Name	SAACHI RAHEJA
Class	D15B
Subject	MAD & PWA Lab
Lab Outcome	LO1: Understand cross platform mobile application development using Flutter framework LO2: Design and Develop interactive Flutter App by using widgets, layouts, gestures and animation LO3: Analyze and Build production ready Flutter App by incorporating backend services and deploying on Android / iOS
Grade:	

5] Single codebase : Write once, run on multiple platforms.

Advantages of Using Flutter:

1] Faster Development Time : Hot reload and a single code base reduce development effort & cost.

2] Cost Effective : Since development, write one code for multiple platforms, it reduces costs associated with maintaining separate teams for iOS & Android.

3] Consistent UI : Flutter renders everything using its own engine, ensuring a uniform look across devices.

4] Growing Ecosystem : Active community and growing library support.

1)b] Discuss how the Flutter Framework differs from traditional approaches? and why it has gained popularity in the developer community?

A] Flutter uses a single codebase for multiple platforms unlike traditional native development that requires separate code for iOS (Swift) & Android (Kotlin). It does not rely on platform specific UI components but instead render everything using its own graphic engine, ensuring consistency. Unlike React

Native, which uses a Javascript bridge, Flutter compiles directly to native ARM code, offering better performance. Its hot reload feature allows developer changes instantly, making development faster & more efficient.

Flutter has gained popularity due to its faster development cost efficiency, & cross platform support. Business prefer it as it reduces development time & costs developing high performance apps. It customizable widget system ensures a smooth, native like experience.

Q] a] Describe the concept of the widget tree in Flutter. Explain the widget composition is used to build complex UI.

A] In flutter everything is a widget (button, text, icon, etc). These widgets are arranged in a hierarchial structure known as the widget tree. The widget tree determines the UI.

Widget composition to build complex UI:

- Flutter encourages a composition-based approach rather than inheritance.
- Instead of creating large, monolithic widget, developer build small, reusable widget that are combined to form complex UIs.

Q.2] b] Provide example of commonly widget & their roles in creating widget.

1] Text : Displays a string of text. It is used to show labels, title and messages.

```
Text ("Hello World!")
```

2] Container : A versatile widget used for creating boxes, including padding, margin, borders, and background color.

```
Container (
    width : 100.0,
    height : 100.0,
    color : Colors.blue,
)
```

3] Rows & Column : Layout widgets for horizontal (Row) or vertical (column) arrangements.

```
Column (
    children : [Text ("Item 1"), Text2 ("Item 2")]
```

4] Scaffold : Provides basic structure such as AppBar, Drawer, NavigationBar & Floating Action Button.

```
Scaffold (
    appBar : AppBar (title : Text "App Title")),
}
```

3] a] Discuss the importance of state management in flutter application.

A] State management is essential in flutter applications because it allows developers to control the changes in the user interface based on user interactions or other events. Without proper state management, an app can become difficult to maintain, and UI updates might be inconsistent or inefficient.

Importance:

1] Consistency: Proper state management ensures that the UI reflects the current state of application, leading to a predictable and consistent user experience.

2] Efficiency: Helps in updating only the necessary parts of the UI when the state changes, rather than rebuilding the entire UI.

3] Scalability: Effective state management makes it easier to manage more complex apps with multiple states and interdependencies.

4] Separation of concerns: It allows the separation of logic from the UI, improving code maintainability and scalability.

3] Install Firebase dependencies & Use FlutterFire plugins to interact with Firebase services.

4] Use Firebase features : Access Firebase services like Firestore, Firebase Authentication, Firebase Storage, etc. by using the appropriate FlutterFire package.

Benefits :-

1] Scalability :- Firebase handles scaling automatically so developers don't have to worry about infrastructure.

2] Real-time Database : Offers real time synchronization of data across users, making it ideal for chat apps.

3] Authentication : Firebase provides built in authentication for email/password, Google, Facebook & other providers.

MAD & PWA Lab Journal

Experiment No.	Assignment-2
Assignment 2 Questions	1. Define Progressive Web App (PWA) and explain its significance in modern web development.Discuss the key characteristics that differentiate PWAs from traditional mobile apps 2. Define responsive web design and explain its importance in the context of Progressive Web Apps.Compare and contrast responsive, fluid, and adaptive web design approaches. 3. Describe the lifecycle of Service Workers, including registration, installation, and activation phases. 4. Explain the use of IndexedDB in the Service Worker for data storage.
Roll No.	46
Name	SAACHI RAHEJA
Class	D15B
Subject	MAD & PWA Lab
Lab Outcome	LO4:Understand various PWA frameworks and their requirements LO5: Design and Develop a responsive User Interface by applying PWA Design techniques LO6:Develop and Analyze PWA Features and deploy it over app hosting solutions
Grade:	

05/05

Name : Saachi Raheja.
Class : D15B
Roll No : 46

Assignment 2 - MPL

1] Define Progressive Web App (PWA) and explain its significance in modern web development. Discuss the key characteristics that differentiate PWAs from traditional mobile app.

A] A progressive Web App (PWA) is a web application that delivers a native app-like experience using modern web technologies. It works across devices, supports offline access, and can be installed without an app store.

Significance in Modern Web Development :

- Cross-Platform : Runs on any device with a web browser.
- Online Support : Uses Service Workers for caching.
- Fast Performance : Loads quickly, even on slow networks.
- Engagement : Supports push notifications and installation.
- Lower Costs : A single codebase reduces development effort.

Feature	PWA	Traditional Mobile App.
Installation	Via browser	App store
Offline Support	Via caching	Via explicit design
Push Notification	Yes	Yes.
Platform dependency	Works on all browsers.	Platform Specific.
Updates	Automatic	Requires user updates

2] Define responsive web design and explain its importance in the context of PWA. Compare and contrast responsive, fluid and adaptive web design approaches.

Responsive Web Design ensures a website adapts dynamically to different screen sizes using flexible layouts and media queries.

Importance in PWAs:

- Consistent User Experience : Optimizes for all screen sizes.
- Better Accessibility : Enhances usability across devices.
- Single Codebase : Reduces development effort.
- SEO Benefits : Google favors mobile friendly design.

Comparison of Web Design Approaches.

Feature	Responsive	Fluid	Adaptive
Definition	Adjusts layout dynamically	Uses percentage based widths	Uses fixed breakpoints
Flexibility	High	High	Limited.
Development Effort	Moderate	Low	High
Best Use Case	PWAs, modern sites	Simple layouts	Specific device layouts.

3] Describe the lifecycle of Service Workers, including registration, installation, and activation phases

A Service Worker is a background script that enhances PWAs by enabling offline caching, push notifications and background sync.

Lifecycle Phases

1] Registration :
- Registered via navigator.serviceWorker.register( ).
- Browser downloads and installs the scripts.

2] Installation
- Triggers the install event
- Caches static assets for offline use

3] Activation
- Triggers the activate event
- Cleans up old caches and takes control of open pages.

4] Idle & Fetch Handling:
- Intercepts network requests and serves cached data.

5] Update & Termination.

- Detects new versions and replaces old workers.
- The browser terminates inactive service workers to save resources.

4] Explain the use of IndexedDB in Service Workers for data Storage.

IndexedDB is a browser based NoSQL database for storing structured data persistently.

Benefits in service workers.

- Offline Storage : Saves data when the user is offline
- Efficient Data Syncing : Updates local data when online.
- Non Blocking Operations : Uses asynchronous API to prevent delays.

Ex:-

```
let db;
const req = indexedDB.open("PWA", 1);

req.onupgradeneeded = function (event) {
  let db = event.target.result;
  db.createObjectStore("articles", {keyPath: "id"});
};

function storeArticle (article) {
  let transaction = db.transaction("articles",
                                   "readwrite");
  let store = transaction.objectStore("articles");
  store.put (article);
}
```