EXPERIMENT No. 1

Name : Arya Manoj Madhavi
Class : D15B
Roll No.: 29

Aim: To Install and configure the flutter environment

Install the Flutter SDK

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 the 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 the 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:\Windows\System32\cmd.e

Microsoft Windows [Version 10.0.26100.2894]
(c) Microsoft Corporation. All rights reserved.

C:\flutter\src\flutter\bin>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.

Available commands:
```

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:\flutter\src\flutter\bin>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 3.27.3, on Microsoft Windows [Version 10.0.26100.2894], locale en-IN)
[✓] Windows Version (Installed version of Windows is version 10 or higher)
[✓] Android toolchain - develop for Android devices (Android SDK version 35.0.1)
[✓] Chrome - develop for the web
[!] Visual Studio - develop Windows apps (Visual Studio Build Tools 2019 16.11.39)
    X The current Visual Studio installation is incomplete.
      Please use Visual Studio Installer to complete the installation or reinstall Visual Studio.
[✓] Android Studio (version 2024.2)
[✓] VS Code (version 1.96.4)
[✓] Connected device (3 available)
[✓] Network resources

! Doctor found issues in 1 category.

C:\flutter\src\flutter\bin>
```

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

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

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

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

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

```
C:\flutter\src\flutter\bin>flutter doctor --android-licenses
Warning: Errors during XML parse:
Warning: Additionally, the fallback loader failed to parse the XML.
Warning: Errors during XML parse:       ] 56% Fetch remote repository...
Warning: Additionally, the fallback loader failed to parse the XML.ry...
[=======================================] 100% Computing updates...
All SDK package licenses accepted.


C:\flutter\src\flutter\bin>
```

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

Step 8.1: To set an Android emulator, go to Android Studio > Tools > Android > AVD Manager and select Create Virtual Device. Or, go to Help->Find Action->Type Emulator in the search box. You will get the following screen.

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

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

Step 8.4: Now, verify the all AVD configuration. If it is correct, click on Finish. The following screen appears.

Step 8.5: Last, click on the icon pointed into the red color rectangle. The Android emulator displayed as below screen.

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

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

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

Step 9.3: Restart the Android Studio.

EXPERIMENT No. 2

Name : Arya Manoj Madhavi
Class : D15B
Roll No.: 29

Aim : To Design Flutter UI by Including Common Widgets

Introduction:
Flutter is an open-source UI software development kit created by Google. It is primarily used to develop applications for Android, iOS, Linux, macOS, Windows, Google Fuchsia, and the web from a single codebase. Flutter's appeal lies in its ability to craft visually stunning user interfaces with smooth animations, coupled with the fact that it uses Dart as its programming language. In this experiment, the focus is on understanding how to design user interfaces by incorporating various common widgets available in Flutter.

Objective:
The aim of this experiment is to design and develop a user-friendly Flutter UI using common widgets such as containers, rows, columns, buttons, text fields, and other essential components. This helps in gaining a clear understanding of how to efficiently utilize these widgets to create interactive and responsive UIs.

Theory:
Flutter revolves around a widget tree, where everything is a widget, from the structural elements like containers and rows to the interactive components such as buttons and text fields. Widgets can be broadly categorized into two types: **Stateful** and **Stateless** widgets.

- **Stateless Widgets**: These widgets are immutable, meaning their properties cannot change during the runtime of the app. Common examples include Text, Icons, and Containers.

- **Stateful Widgets**: These widgets maintain state, meaning their properties can change dynamically during the app’s lifecycle. Examples include Checkbox, Switch, and Slider.

Some of the most commonly used widgets in Flutter include:

- **Container**: A flexible widget used for layout control, it can contain other widgets and control the padding, alignment, and size.

- **Row and Column**: These widgets are used for arranging child widgets in a horizontal (Row) or vertical (Column) direction.

- **Text**: This widget displays a string of text and can be customized with various properties like font size, style, and alignment.

- **Buttons**: Buttons like ElevatedButton, TextButton, and IconButton are used to handle user interactions.

- **Image**: Displays an image, which can be loaded from assets or the internet.

- **TextField**: Allows users to input text, a key component for forms.

By combining these widgets, developers can design UIs that are both aesthetic and functional. Flutter's rich collection of pre-built widgets provides great flexibility in UI design, enabling developers to create complex interfaces with ease.

Code:

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

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

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

class HomePage extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(
       title: Text("Expense Tracker"),
       backgroundColor: Colors.brown,
     ),
     body: Column(
       children: [
         // Budget Left Section
         Container(
           padding: EdgeInsets.all(16.0),
           decoration: BoxDecoration(
             color: Colors.brown.shade200,
             borderRadius: BorderRadius.circular(10),
```

```
          ),
          margin: EdgeInsets.all(16.0),
          child: Column(
            crossAxisAlignment: CrossAxisAlignment.start,
            children: [
              Text("Monthly Budget Left", style: TextStyle(fontSize: 18, fontWeight:
FontWeight.bold)),
              SizedBox(height: 8),
              Text("₹25,000", style: TextStyle(fontSize: 22, fontWeight: FontWeight.bold, color:
Colors.red)),
            ],
          ),
        ),

        // Expense Sections
        Expanded(
          child: ListView(
            padding: EdgeInsets.all(16.0),
            children: [
              TransactionItem("Food", "₹500", Colors.red),
              TransactionItem("Transport", "₹150", Colors.blue),
              TransactionItem("Shopping", "₹1200", Colors.purple),
              TransactionItem("Entertainment", "₹800", Colors.orange),
            ],
          ),
        ),

        // Buttons for Adding Expense & Income
        Padding(
          padding: const EdgeInsets.all(16.0),
          child: Row(
            mainAxisAlignment: MainAxisAlignment.spaceEvenly,
            children: [
              ElevatedButton(
                onPressed: () {},
                style: ElevatedButton.styleFrom(backgroundColor: Colors.red),
                child: Text("Add Expense"),
              ),
              ElevatedButton(
                onPressed: () {},
                style: ElevatedButton.styleFrom(backgroundColor: Colors.green),
                child: Text("Add Income"),
              ),
            ],
```

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

class TransactionItem extends StatelessWidget {
 final String category;
 final String amount;
 final Color color;

 TransactionItem(this.category, this.amount, this.color);

 @override
 Widget build(BuildContext context) {
   return Container(
     margin: EdgeInsets.symmetric(vertical: 8.0),
     padding: EdgeInsets.all(16.0),
     decoration: BoxDecoration(
       color: color.withOpacity(0.1),
       borderRadius: BorderRadius.circular(10),
     ),
     child: Row(
       mainAxisAlignment: MainAxisAlignment.spaceBetween,
       children: [
         Text(category, style: TextStyle(fontSize: 16, fontWeight: FontWeight.bold)),
         Text(amount, style: TextStyle(fontSize: 16, fontWeight: FontWeight.bold, color:
Colors.red)),
       ],
     ),
   );
 }
}
```

OUTPUT:

Conclusion:

Designing a Flutter UI involves an understanding of the different common widgets and how they can be organized within the widget tree. By using widgets like Container, Row, Column, Text, Buttons, and TextField, developers can create responsive, interactive, and visually appealing UIs. This experiment helps in mastering the fundamental components of Flutter UI design, providing the basis for building more complex user interfaces in future projects.

EXPERIMENT No. 3

Name : Arya Manoj Madhavi
Class : D15B
Roll No.: 29

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

Objective:

To understand the process of integrating icons, images, and custom fonts in a Flutter application, which are essential components in building a visually engaging and user-friendly interface.

Introduction:

Flutter, a popular open-source UI framework, is known for its flexibility and ease of building cross-platform applications with a single codebase. It allows developers to create high-performance apps for mobile, web, and desktop. One of the key features of Flutter is its ability to include various UI elements such as icons, images, and custom fonts, which significantly enhance the overall user experience and improve the application's aesthetic appeal. By integrating these elements, developers can align the application's design with modern UI/UX standards, improving usability and visual coherence.

Theory:

1. Icons in Flutter: Icons are crucial in communicating functionality and guiding users through the app interface. Flutter comes with a rich set of Material Design icons that are easy to implement. Additionally, it allows for the integration of custom icons from third-party libraries such as Font Awesome and Ionicons.
   Icons can also be customized in terms of size, color, and positioning to match the overall theme and design of the app. The ability to use both built-in and custom icons offers great flexibility for designers to ensure consistency with branding and user preferences. Custom icons can be created as part of an icon font or as individual image files. Proper usage of icons can reduce the cognitive load on users, making navigation simpler and more intuitive.
2. Images in Flutter: Images are essential to create engaging, context-rich apps. Flutter allows the inclusion of images from multiple sources:
   - Asset Images (Local Storage): These are images bundled with the application itself. They are stored in the app’s project directory under the "assets" folder and referenced in the Flutter project configuration file (pubspec.yaml). Asset images are especially useful for app icons, logos, splash screens, and other elements that are static and always present in the app.
   - Network Images (URL-based): Flutter supports loading images directly from URLs, making it easy to integrate dynamic content, such as product images

fetched from a web API. This is particularly useful for applications that display frequently updated content from remote sources, such as e-commerce apps, news apps, or social media platforms.
    - Memory Images: For more advanced use cases, images can be decoded and displayed directly from memory. This allows for the integration of images generated or processed at runtime, for example, images captured by the user's camera or received through real-time communication channels.
3. Images can also be styled, resized, and transformed (e.g., rotated or scaled) within Flutter using the Image widget. This flexibility enables developers to optimize the layout and visual flow of their applications, ensuring that images appear sharp and well-positioned across various screen sizes.
4. Fonts in Flutter: Custom fonts are an important tool for creating visually distinctive applications. By using custom fonts, designers can align the app's typography with the brand's identity or simply enhance the readability and style of the text. Flutter provides support for custom fonts through two main methods:
    - Google Fonts Integration: Flutter offers a convenient way to access the extensive library of Google Fonts, making it easy to integrate modern, open-source fonts into the app. Google Fonts can be added via a simple package, offering a wide selection without the need to manually download or store font files.
    - Manually Added Fonts: Developers can also add their own fonts, such as .ttf or .otf files, to the app's asset folder. These custom fonts must be declared in the pubspec.yaml file, ensuring that Flutter can access and use them in the application. Custom fonts provide a unique touch to an app's design, allowing for a tailored user experience that reflects the app's theme or purpose.
5. Additionally, Flutter allows for advanced typography features, such as setting specific fonts for different parts of the app (e.g., using a different font for headers and body text), adjusting font weight, size, letter spacing, and text alignment. This level of control ensures that developers can maintain a consistent visual identity across the application.

Steps:

1. Set up icons:
    - Use Flutter's built-in Material Icons or import third-party icon libraries to expand the icon set available in your app. Customize the appearance of icons to match the app's theme.
2. Add images:
    - Store images in the assets folder and reference them in the app configuration (pubspec.yaml).
    - Use network images to display dynamic content fetched from the web.
    - Customize the display, layout, and resizing of images based on the app's design needs.
3. Integrate custom fonts:
    - Download and add font files (or use Google Fonts) and declare them in the pubspec.yaml file.

    - Apply the custom fonts to text widgets to improve the visual appeal and branding of the app.

Best Practices:

- Organize Assets Efficiently: Place assets like icons, images, and fonts in well-structured directories and ensure they are properly referenced in pubspec.yaml. Maintaining an organized asset folder structure helps prevent errors and makes the project more maintainable.
- Optimize Image Sizes: Use appropriately sized images to prevent unnecessary memory usage and loading time. Images should be optimized for different device resolutions and screen densities to maintain high quality without slowing down the app.
- Fallback Fonts: When using custom fonts, provide a fallback font to avoid rendering issues in case the custom font fails to load or is unsupported on certain devices.
- Testing on Multiple Devices: It’s important to test the app on various screen sizes and platforms to ensure that icons, images, and fonts are rendered correctly and maintain their quality across devices.

Code:-

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

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

class ExpenseTrackerApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     debugShowCheckedModeBanner: false,
     title: 'Expense Tracker',
     theme: ThemeData(
       primarySwatch: Colors.blue,
       fontFamily: 'Roboto', // Custom Font
     ),
     home: HomePage(),
   );
 }
}

class HomePage extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return Scaffold(
```

```
appBar: AppBar(
  title: Text('Expense Tracker'),
  centerTitle: true,
),
body: Center(
  child: Column(
    mainAxisAlignment: MainAxisAlignment.center,
    children: [
      // Using Font Awesome Icon
      Icon(FontAwesomeIcons.dollarSign, size: 50, color: Colors.green),
      SizedBox(height: 20),
      // Using Built-in Icon
      Icon(Icons.pie_chart, size: 50, color: Colors.blue),
      SizedBox(height: 20),
      // Displaying Image
      Image.asset(
        'assets/images/logo.png',
        width: 150,
        height: 150,
      ),
      SizedBox(height: 20),
      // Using Custom Font
      Text(
        'Track Your Expenses',
        style: TextStyle(fontSize: 24, color: Colors.black),
      ),
    ],
  ),
),
bottomNavigationBar: BottomNavigationBar(
  items: const [
    BottomNavigationBarItem(
      icon: Icon(Icons.home),
      label: 'Home',
    ),
    BottomNavigationBarItem(
      icon: Icon(Icons.pie_chart),
      label: 'Statistics',
    ),
    BottomNavigationBarItem(
      icon: Icon(FontAwesomeIcons.user),
      label: 'Profile',
    ),
  ],
```

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

pubspec.yaml

```
assets:
 - assets/images/logo.png

fonts:
 - family: Roboto
   fonts:
     - asset: assets/fonts/Roboto-Regular.ttf
```

Output:

Conclusion:

Integrating icons, images, and fonts into a Flutter application is essential for creating visually engaging and user-friendly interfaces. Proper integration allows for consistent branding, enhanced UI aesthetics, and improved user experience. However, careful attention must be given to asset management, file paths, and project configuration. Misconfigurations in the pubspec.yaml file or incorrect asset paths can lead to common issues such as "asset not found" or "font loading errors." Through proper testing and management, these issues can be mitigated, resulting in a smooth, visually appealing application with rich multimedia content.

EXPERIMENT No. 4

Name : Arya Manoj Madhavi
Class : D15B
Roll No.: 29

Aim : To create an interactive Form using form widget

Objective:
To understand the process of building interactive forms in Flutter using the Form widget. Forms are essential components of any application that involves user input, such as login screens, registration pages, and feedback forms. This experiment aims to explore how to efficiently create, validate, and manage form inputs to provide a seamless user experience.

Introduction:
Flutter, a powerful UI framework for building cross-platform applications, offers a comprehensive suite of widgets to create interactive user interfaces. One such essential widget is the Form widget, which enables developers to create and manage forms efficiently. By utilizing Form, TextFormField, and other related widgets, developers can easily handle user inputs, perform validations, and manage form states.
Creating interactive forms in Flutter is not just about capturing user data but also about enhancing the user experience through intuitive designs, real-time feedback, and efficient error handling. Properly designed forms contribute to higher user engagement and better data accuracy. This experiment will demonstrate how to implement an interactive form with input validation and state management in Flutter.

Theory:

Form Widget in Flutter:
The Form widget in Flutter acts as a container for form fields like TextFormField, DropdownButton, and more. It provides a convenient way to group input fields, manage their state, and perform validations. Forms in Flutter are typically used in conjunction with a GlobalKey to access form states, such as validating input fields or saving the form's data.
The Form widget offers the following key features:

- State Management: Using GlobalKey<FormState>, developers can manage the state of form fields, check for validation errors, and save the inputs.
- Validation: The Form widget supports custom validation logic, ensuring data accuracy and completeness before submission.
- Input Handling: It allows easy integration with various input fields, including text, numbers, dates, and dropdowns.

TextFormField in Flutter:
TextFormField is a commonly used input widget within a Form. It provides built-in features like:

- Input Decoration: Customizable labels, icons, and hints for better UX.

- Validation: Using the validator property, custom validation rules can be implemented.
- Keyboard Type: The keyboard layout can be adjusted (e.g., numeric, email, text) based on the input requirement.
- Controllers: TextEditingController manages the text entered, allowing for easy retrieval and manipulation of user inputs.

Form Validation:
Validating user input is crucial for maintaining data integrity and enhancing user experience. Flutter allows developers to:

- Implement custom validation logic using the validator property.
- Display error messages for invalid inputs.
- Use the FormState's validate() method to check the form's validity before submission.

Form Submission:
Form submission is triggered after validating all inputs. Using FormState's save() method, developers can collect and process user data efficiently. This can involve sending data to a server, storing it locally, or performing other actions.
Steps:

1. Set up Form Widget:
    - Create a Form widget and associate it with a GlobalKey<FormState> to manage the state of the form fields.
    - Wrap the form in a SingleChildScrollView to ensure it's scrollable on smaller screens.
2. Add Input Fields:
    - Use TextFormField widgets for text inputs such as Name, Email, and Password.
    - Customize input fields with labels, hints, and icons for better UX.
    - Use DropdownButtonFormField for selection inputs (e.g., Gender, Country).
3. Implement Validation:
    - Add validation logic to each TextFormField using the validator property.
    - Display error messages for invalid inputs, such as empty fields or incorrect email formats.
    - Use FormState's validate() method to ensure all fields are valid before submission.
4. Submit Form Data:
    - Use ElevatedButton for the form submission button.
    - On pressing the submit button, validate all inputs.
    - If the form is valid, use FormState's save() method to collect input data.
    - Display a success message or navigate to another screen upon successful submission.

Code:-

Signup page

```
// lib/screens/signup_screen.dart
import 'package:flutter/material.dart';
import 'package:shared_preferences/shared_preferences.dart';
class SignupScreen extends StatefulWidget {
 @override
 _SignupScreenState createState() => _SignupScreenState();
}
class _SignupScreenState extends State<SignupScreen> {
 final TextEditingController nameController = TextEditingController();
 final TextEditingController emailController = TextEditingController();
 final TextEditingController passwordController = TextEditingController();
 final _formKey = GlobalKey<FormState>();
 bool isLoading = false;
 Future<void> _signup() async {
   if (_formKey.currentState!.validate()) {
     setState(() {
       isLoading = true;
     });
     try {
       print("Starting signup...");
       await Future.delayed(Duration(seconds: 2)); // Simulate network delay
       // Save User Data Locally
       SharedPreferences prefs = await SharedPreferences.getInstance();
       await prefs.setString('userName', nameController.text);
       await prefs.setString('userEmail', emailController.text);
       print("Signup successful. Navigating to profile...");
       setState(() {
         isLoading = false;
       });
       // Navigate to Home Screen
       Navigator.pushReplacementNamed(context, '/home');
     } catch (e) {
       setState(() {
         isLoading = false;
       });
       print('Error: $e');
     }
   }
 }
 @override
 Widget build(BuildContext context) {
   return Scaffold(
     backgroundColor: Colors.green[50],
     body: Center(
       child: SingleChildScrollView(
         child: Padding(
           padding: const EdgeInsets.all(16.0),
           child: Form(
```

```
key: _formKey,
child: Card(
  shape: RoundedRectangleBorder(
    borderRadius: BorderRadius.circular(15.0),
  ),
  elevation: 8,
  child: Padding(
    padding: const EdgeInsets.all(20.0),
    child: Column(
      crossAxisAlignment: CrossAxisAlignment.stretch,
      children: [
        Text(
          'Sign Up',
          style: TextStyle(
            fontSize: 30,
            fontWeight: FontWeight.bold,
            color: Colors.green,
          ),
          textAlign: TextAlign.center,
        ),
        SizedBox(height: 20),
        TextFormField(
          controller: nameController,
          decoration: InputDecoration(
            labelText: 'Name',
            border: OutlineInputBorder(),
            prefixIcon: Icon(Icons.person),
          ),
          validator: (value) {
            if (value == null || value.isEmpty) {
              return 'Please enter your name';
            }
            return null;
          },
        ),
        SizedBox(height: 15),
        TextFormField(
          controller: emailController,
          decoration: InputDecoration(
            labelText: 'Email',
            border: OutlineInputBorder(),
            prefixIcon: Icon(Icons.email),
          ),
          keyboardType: TextInputType.emailAddress,
          validator: (value) {
            if (value == null || value.isEmpty) {
              return 'Please enter your email';
            }
            if (!RegExp(r'^[^@]+@[^@]+\.[^@]+').hasMatch(value)) {
```

```
            return 'Enter a valid email';
          }
          return null;
        },
      ),
      SizedBox(height: 15),
      TextFormField(
        controller: passwordController,
        decoration: InputDecoration(
          labelText: 'Password',
          border: OutlineInputBorder(),
          prefixIcon: Icon(Icons.lock),
        ),
        obscureText: true,
        validator: (value) {
          if (value == null || value.isEmpty) {
            return 'Please enter your password';
          }
          if (value.length < 6) {
            return 'Password must be at least 6 characters';
          }
          return null;
        },
      ),
      SizedBox(height: 20),
      isLoading
          ? Center(child: CircularProgressIndicator())
          : ElevatedButton(
        onPressed: _signup,
        child: Text('Sign Up'),
        style: ElevatedButton.styleFrom(
          backgroundColor: Colors.green,
          shape: RoundedRectangleBorder(
            borderRadius: BorderRadius.circular(10),
          ),
          padding: EdgeInsets.all(15),
        ),
      ),
      TextButton(
        onPressed: () {
          Navigator.pushNamed(context, '/login');
        },
        child: Text('Already have an account? Login'),
      ),
    ],
  ),
),
),
),
```

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

Profile page

```
// lib/screens/profile_screen.dart
import 'package:flutter/material.dart';
import 'package:shared_preferences/shared_preferences.dart';
class ProfileScreen extends StatefulWidget {
 @override
 _ProfileScreenState createState() => _ProfileScreenState();
}
class _ProfileScreenState extends State<ProfileScreen> {
 String userName = "User Name";
 String userEmail = "user@example.com";

 @override
 void initState() {
  super.initState();
  _loadUserData();
 }
 Future<void> _loadUserData() async {
  SharedPreferences prefs = await SharedPreferences.getInstance();
  setState(() {
    userName = prefs.getString('userName') ?? "User Name";
    userEmail = prefs.getString('userEmail') ?? "user@example.com";
  });
 }
 Future<void> _logout() async {
  SharedPreferences prefs = await SharedPreferences.getInstance();
  await prefs.clear();
  Navigator.pushReplacementNamed(context, '/login');
 }
 @override
 Widget build(BuildContext context) {
  return Scaffold(
    backgroundColor: Colors.blue[50],
    body: Center(
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          CircleAvatar(
            radius: 50,
            child: Icon(
              Icons.person,
              size: 50,
```

```
            color: Colors.green[800],
          ),
        ),
        SizedBox(height: 20),
        Text(
          userName,
          style: TextStyle(fontSize: 24, fontWeight: FontWeight.bold),
        ),
        SizedBox(height: 10),
        Text(
          userEmail,
          style: TextStyle(fontSize: 16, color: Colors.grey),
        ),
        SizedBox(height: 20),
        ElevatedButton(
          onPressed: _logout,
          child: Text('Logout'),
          style: ElevatedButton.styleFrom(
            backgroundColor: Colors.redAccent,
            shape: RoundedRectangleBorder(
              borderRadius: BorderRadius.circular(10),
            ),
          ),
        ),
      ],
    ),
  ),
);
}
}
```

Output:

Conclusion:

Creating an interactive form using the Form widget in Flutter is crucial for capturing user input efficiently and enhancing the overall user experience. Proper implementation ensures data accuracy, smooth validation, and seamless state management. However, attention must be given to input handling, validation logic, and responsive design. Misconfigurations in the form's state management or inadequate validation rules can lead to issues such as form submission errors or inconsistent user feedback. By following best practices and thoroughly testing on various devices, these issues can be minimized, resulting in a reliable, user-friendly form that enhances usability and engagement.

EXPERIMENT No. 5

Name : Arya Manoj Madhavi
Class : D15B
Roll No.: 29

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

Objective:
To understand the process of applying navigation, routing, and gestures in a Flutter application. These elements are fundamental for creating interactive and user-friendly apps, enabling users to move between different screens, pages, or components seamlessly. This experiment aims to explore how to efficiently implement navigation, manage routes, and integrate gesture controls to enhance the overall user experience.

Introduction:
Flutter, a versatile UI framework for cross-platform application development, provides powerful tools for navigation, routing, and gesture recognition. Efficient navigation ensures smooth transitions between screens, improving usability and user engagement. Flutter's Navigator widget and routing system offer various navigation techniques, including named routes, anonymous routes, and dynamic routing.
Gestures, on the other hand, enable users to interact with the app using taps, swipes, pinches, and other touch inputs. By leveraging Flutter's GestureDetector widget and pre-built gesture widgets, developers can create intuitive and interactive user experiences. This experiment will demonstrate how to implement navigation, routing, and gestures effectively in a Flutter application.

Theory:

Navigation in Flutter:
Navigation in Flutter is handled using the Navigator widget, which manages a stack of route objects. Routes represent different screens or pages within the application. Flutter provides multiple ways to navigate between these routes:

- Anonymous Routes: Directly push and pop routes without naming them, suitable for simple navigation flows.
- Named Routes: Define routes with specific names in the MaterialApp widget, allowing for organized and maintainable navigation.
- Dynamic Routing: Utilize onGenerateRoute for flexible route management and advanced navigation scenarios.

Flutter's navigation system supports various transitions, such as slide, fade, or custom animations, enhancing the user experience with smooth screen transitions.

Routing in Flutter:
Routing is the mechanism of mapping paths or URLs to specific screens. Flutter supports:

- Static Routing: Defined in the MaterialApp's routes property for fixed navigation paths.
- Dynamic Routing: Managed through onGenerateRoute or onUnknownRoute for more flexible routing scenarios, such as deep linking or dynamic URL handling.

Proper routing management ensures efficient navigation flow and improves app maintainability by keeping route definitions centralized and organized.

Gestures in Flutter:
Gestures are an essential aspect of interactive app design, enabling users to engage with the app through touch inputs. Flutter provides the GestureDetector widget to capture various gestures, including:

- Tap: Single or double taps for actions like button clicks.
- Swipe: Horizontal or vertical swipes for navigation or item dismissal.
- Long Press: Used for context menus or additional options.
- Pan and Drag: For draggable UI elements or custom animations.

Flutter also offers pre-built gesture widgets like InkWell and GestureDetector, which provide ripple effects and custom gesture handling, enhancing interactivity and user experience.

Steps:

1. Set up Navigation:
    - Use MaterialApp as the root widget and define routes in the routes property.
    - Implement both named and anonymous routes for flexible navigation patterns.
    - Use Navigator.push() and Navigator.pop() for stack-based navigation.
    - Apply Navigator.pushReplacement() or Navigator.pushNamedAndRemoveUntil() for controlled navigation flows.
2. Implement Routing:
    - Define static routes in MaterialApp for commonly used paths (e.g., Home, About, Contact).
    - Use onGenerateRoute for dynamic routing, supporting scenarios like passing parameters between screens.
    - Apply onUnknownRoute for custom 404 pages or error handling.
3. Add Gestures:
    - Use GestureDetector for custom gesture recognition (e.g., tap, double-tap, swipe).
    - Implement InkWell for material ripple effects and standard touch interactions.
    - Add swipe gestures for navigation (e.g., swiping between screens or dismissing items).
    - Integrate advanced gestures like drag and drop for enhanced interactivity.
4. Customize Navigation Transitions:
    - Implement custom page transitions using PageRouteBuilder.
    - Add animations like slide, fade, or scale for smooth navigation experiences.
    - Customize transition duration and curves for a polished UI effect.

Best Practices:

- Organize Routes: Centralize route definitions for maintainability and better code organization.
- Consistent Navigation Flow: Maintain a consistent navigation pattern for better usability.
- Accessibility Considerations: Ensure gestures are intuitive and accessible for all users.
- Testing on Multiple Devices: Verify navigation and gesture performance on various screen sizes and platforms.

Code:-

```
import 'package:flutter/material.dart';
import 'screens/auth/login_screen.dart';
import 'screens/auth/signup_screen.dart';
import 'screens/home_screen.dart';
import 'screens/add_expense_screen.dart';
import 'screens/add_income_screen.dart';
import 'screens/calendar_screen.dart';
import 'screens/statistics_screen.dart';
import 'screens/profile_screen.dart';

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

class ExpenseTrackerApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Expense Tracker',
     theme: ThemeData(primarySwatch: Colors.blue),
     initialRoute: '/signup',
     routes: {
       '/login': (context) => LoginScreen(),
       '/signup': (context) => SignupScreen(),
       '/home': (context) => HomeScreen(),
       // '/add-expense': (context) => AddExpenseScreen(),
       // '/add-income': (context) => AddIncomeScreen(),
       // '/calendar': (context) => CalendarScreen(),
       // '/statistics': (context) => StatisticsScreen(),
       '/profile': (context) => ProfileScreen(),
     },
   );
 }
}
```

Navigation

```
Future<void> _logout() async {
 SharedPreferences prefs = await SharedPreferences.getInstance();
 await prefs.clear();
 Navigator.pushReplacementNamed(context, '/login');
}
```

Gestures

```
import 'package:flutter/material.dart';
import 'package:flutter_calendar_carousel/flutter_calendar_carousel.dart'
   show CalendarCarousel;
import 'package:intl/intl.dart';

class CalendarScreen extends StatefulWidget {
 final List<Map<String, dynamic>> expenses;

 CalendarScreen({required this.expenses});

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

class _CalendarScreenState extends State<CalendarScreen> {
 DateTime? _selectedDate;
 List<Map<String, dynamic>> _selectedDateExpenses = [];

 // Function to filter expenses for the selected date
 void _filterExpensesByDate(DateTime date) {
   setState(() {
     _selectedDate = date;
     String formattedDate = DateFormat('yyyy-MM-dd').format(date);
     _selectedDateExpenses = widget.expenses.where((expense) {
       return expense['date'] == formattedDate;
     }).toList();
   });
 }

 @override
 Widget build(BuildContext context) {
   return Scaffold(
     appBar: AppBar(title: Text('Expense Calendar')),
     body: Column(
       children: [
         CalendarCarousel(
```

```
  onDayPressed: (date, events) {
    _filterExpensesByDate(date);
  },
  weekendTextStyle: TextStyle(color: Colors.red),
  thisMonthDayBorderColor: Colors.grey,
  weekFormat: false,
  height: 420.0,
  selectedDateTime: _selectedDate,
  daysHaveCircularBorder: false,
),
SizedBox(height: 10),
// Display expenses for the selected date with Scrollbar
Expanded(
  child: Scrollbar(
    thumbVisibility: true,
    thickness: 8,
    radius: Radius.circular(8),
    child: _selectedDateExpenses.isNotEmpty
        ? ListView.builder(
      itemCount: _selectedDateExpenses.length,
      itemBuilder: (context, index) {
        var expense = _selectedDateExpenses[index];
        return Card(
          color: Colors.pink[50],
          child: ListTile(
            title: Text(expense['category']),
            subtitle: Text("Date: ${expense['date']}"),
            trailing: Text(
              "₹${expense['amount'].toStringAsFixed(2)}",
              style: TextStyle(
                  color: Colors.redAccent,
                  fontWeight: FontWeight.bold),
            ),
          ),
        );
      },
    )
        : Center(
      child: Text(
        "No expenses for this date",
        style: TextStyle(fontSize: 18, color: Colors.grey),
      ),
    ),
  ),
```

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

Output:

Conclusion:

Applying navigation, routing, and gestures in Flutter is crucial for building intuitive and interactive applications. Proper implementation ensures seamless user flow, organized route management, and engaging touch interactions. However, attention must be given to state management, route organization, and gesture handling to avoid issues like navigation loops, unexpected gestures, or inconsistent UI behavior. By following best practices and thorough testing, these challenges can be minimized, resulting in a smooth, user-friendly application with dynamic navigation and responsive gestures.

EXPERIMENT No. 6

Name : Arya Manoj Madhavi
Class : D15B
Roll No.: 29

Aim : To Connect Flutter UI with fireBase database

Objective:
To understand the process of connecting a Flutter UI with a Firebase database. Integrating Firebase with Flutter allows developers to build dynamic and responsive applications with real-time data synchronization, user authentication, and cloud storage. This experiment aims to explore how to efficiently connect Flutter UI components to Firebase, perform CRUD (Create, Read, Update, Delete) operations, and manage real-time data updates to provide an interactive user experience.

Introduction:
Flutter, a powerful cross-platform UI framework, combined with Firebase, a robust backend-as-a-service platform, offers a seamless way to build cloud-connected mobile applications. Firebase provides several services like Firestore (NoSQL database), Realtime Database, Authentication, Cloud Storage, and more. By integrating Firebase with Flutter, developers can create feature-rich apps with real-time data updates, user authentication, and cloud-hosted content.
This experiment will demonstrate how to connect Flutter UI components to Firebase, perform CRUD operations, and manage real-time data synchronization. Additionally, it will cover how to handle authentication, error handling, and data security, ensuring a robust and scalable application architecture.

Theory:

Firebase Integration in Flutter:
Firebase offers multiple services that can be integrated with Flutter, including:

- Firestore: A NoSQL cloud database that stores and syncs data in real time across users and devices.
- Realtime Database: A cloud-hosted NoSQL database supporting real-time data syncing.
- Firebase Authentication: Simplifies user authentication with ready-to-use sign-in methods, including email/password and social logins.
- Cloud Storage: Provides scalable storage solutions for media and user-generated content.

To connect Flutter with Firebase, the Firebase SDK needs to be configured in the Flutter project, followed by dependency setup in the pubspec.yaml file. This includes adding necessary packages like firebase_core, cloud_firestore, firebase_auth, and more, depending on the app's requirements.

Firestore and Realtime Database:

- Firestore: A document-oriented database storing data as collections and documents, enabling flexible data models and complex queries. It supports real-time data synchronization, offline capabilities, and powerful querying options, including compound queries and indexing.
- Realtime Database: Stores data as JSON trees, suitable for simple data models and syncing data in real-time across connected clients. It provides event listeners to detect changes in data and trigger UI updates.

CRUD Operations:

CRUD (Create, Read, Update, Delete) operations are fundamental for interacting with Firebase databases:

- Create: Adding new documents or nodes to Firestore or Realtime Database.
- Read: Fetching data using queries, streams, or listeners for real-time updates.
- Update: Modifying existing data without overwriting other fields.
- Delete: Removing documents or nodes from the database.

Authentication and Security:

Firebase Authentication simplifies user sign-in with multiple methods:

- Email/Password Authentication: Traditional sign-in method using email and password.
- Social Logins: Google, Facebook, and other third-party authentication providers.
- Anonymous Authentication: Allows users to explore the app without signing in.

Security rules in Firebase ensure data integrity and control access based on user roles or authentication status. Properly configuring these rules prevents unauthorized data access or manipulation.

Steps:

1. **Firebase Setup and Configuration:**
    - Create a Firebase project and add the Flutter app to the project.
    - Download the google-services.json (for Android) and GoogleService-Info.plist (for iOS) and add them to the respective directories in the Flutter project.
    - Add Firebase dependencies to the pubspec.yaml file, including firebase_core, cloud_firestore, and firebase_auth.
    - Initialize Firebase in the main.dart file using Firebase.initializeApp().
2. **Connecting Flutter UI to Firebase:**
    - Implement StreamBuilder or FutureBuilder to connect Flutter UI components with Firebase data.
    - Use Firestore queries or Realtime Database references to fetch data and update the UI in real-time.
    - Display dynamic data using ListView.builder() or GridView.builder() for efficient rendering.

3. **Performing CRUD Operations:**
    - Create: Add new documents or nodes using add() or set() methods.
    - Read: Fetch data using get() for one-time reads or snapshots() for real-time updates.
    - Update: Use update() to modify specific fields without overwriting other data.
    - Delete: Remove documents or nodes using the delete() method.
4. **Implementing Authentication:**
    - Integrate Firebase Authentication for user sign-in and registration.
    - Implement email/password authentication with validation and error handling.
    - Add Google sign-in using the firebase_auth and google_sign_in packages.
    - Manage authentication states using StreamBuilder to display the appropriate UI for signed-in or guest users.
5. **Error Handling and Security:**
    - Handle errors gracefully, such as network failures, authentication issues, or permission denials.
    - Display user-friendly error messages using Flutter's SnackBar or Dialog widgets.
    - Configure Firebase security rules to restrict unauthorized access and ensure data integrity.
6. **Real-Time Updates and State Management:**
    - Use StreamBuilder to listen to real-time updates from Firestore or Realtime Database.
    - Implement state management using Provider, Riverpod, or other state management solutions for better performance and maintainability.

Best Practices:

- Organize Data Structure: Design Firestore or Realtime Database structure efficiently for scalability and performance.
- Error Handling: Implement comprehensive error handling to enhance user experience and maintain app stability.
- Optimize Performance: Use pagination and indexing for efficient data fetching and reduced loading times.
- Security Rules: Set up Firebase security rules to control data access and prevent unauthorized modifications.
- State Management: Apply appropriate state management techniques (e.g., Provider, Riverpod) for maintainable code architecture.

Code:-

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

import 'screens/auth/login_screen.dart';
import 'screens/auth/signup_screen.dart';
import 'screens/home_screen.dart';
import 'screens/add_expense_screen.dart';
import 'screens/add_income_screen.dart';
import 'screens/calendar_screen.dart';
import 'screens/statistics_screen.dart';
import 'screens/profile_screen.dart';

void main() async {
 WidgetsFlutterBinding.ensureInitialized();
 await Firebase.initializeApp(
   options: DefaultFirebaseOptions.currentPlatform,
 );

 runApp(ExpenseTrackerApp());
}

class ExpenseTrackerApp extends StatelessWidget {
 @override
 Widget build(BuildContext context) {
   return MaterialApp(
     title: 'Expense Tracker',
     theme: ThemeData(primarySwatch: Colors.blue),
     initialRoute: '/signup',
     routes: {
       '/login': (context) => LoginScreen(),
       '/signup': (context) => SignupScreen(),
       '/home': (context) => HomeScreen(),
       // '/add-expense': (context) => AddExpenseScreen(),
       // '/add-income': (context) => AddIncomeScreen(),
       // '/calendar': (context) => CalendarScreen(),
       // '/statistics': (context) => StatisticsScreen(),
       '/profile': (context) => ProfileScreen(),
     },
   );
 }
}
```

Firebase_options_file:

```
import 'package:firebase_core/firebase_core.dart';
class DefaultFirebaseOptions {
 static FirebaseOptions get currentPlatform {
   // Web configuration
   return const FirebaseOptions(
       apiKey: "apikey",
       authDomain: "authdomain",
       projectId: "projectid",
       storageBucket: "",
       messagingSenderId: "",
       appId: "",
       measurementId: ""
// Initialize Firebase
   );
 }
}
```

Output:

Conclusion:
Connecting Flutter UI with Firebase is crucial for building dynamic, cloud-connected applications. Proper integration allows seamless data synchronization, efficient CRUD operations, and secure user authentication. However, challenges like network issues, security misconfigurations, or performance bottlenecks may arise. By following best practices, thorough testing, and strategic data structuring, these challenges can be minimized, resulting in a responsive and robust application with real-time data updates and cloud functionalities.

EXPERIMENT No. 7

Name : Arya Manoj Madhavi
Class : D15B
Roll No.: 29

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

Theory:

Making a Vite React App a Progressive Web App (PWA)

A Progressive Web App (PWA) is a type of web application that provides a reliable, fast, and engaging user experience similar to a native mobile app. It leverages modern web technologies, including service workers, caching strategies, and a web app manifest, to enable offline functionality, push notifications, and an installable experience.

In the context of a Vite React application, converting the app into a PWA involves integrating these key components:

1. Service Workers:
    - Service workers act as a proxy between the browser and the network, enabling background processes such as caching and offline capabilities.
    - They allow the app to load even when there is no internet connection by storing static assets locally.
2. Web App Manifest:
    - The `manifest.json` file provides metadata about the app, including its name, icons, theme color, and display mode.
    - This file helps the browser recognize the app as installable and controls its behavior when launched.
3. Vite PWA Plugin:
    - Since Vite does not have built-in PWA support, a plugin like `vite-plugin-pwa` is required to handle service worker registration and manifest integration automatically.
    - This plugin simplifies the process of enabling caching, auto-updates, and background synchronization.

Key Features of a PWA:

- Offline Availability: Allows users to access the app without an internet connection.

- Fast Performance: Uses caching strategies to reduce loading time.
- App-Like Interface: Can be installed on devices and launched in a standalone mode.
- Secure & Reliable: Served over HTTPS to prevent data tampering.
- Background Sync & Push Notifications: Keeps users engaged with real-time updates.

Steps to Make Your Vite React App a PWA

1 Install Vite PWA Plugin

1. Open your terminal inside the project folder.
2. Run the command to install the PWA plugin.
3. This plugin helps to add service workers and caching automatically.

2 Configure Vite for PWA

1. Open the `vite.config.js` file in your project.
2. Import and add the PWA plugin inside the `plugins` section.
3. Set up options like auto-updates, caching, and offline support.
4. Save the file and restart the Vite server.

3 Add `manifest.json` File

1. Go to the `public` folder in your project.
2. Create a new file named `manifest.json`.
3. Add details like the app name, icons, theme color, and display mode.
4. This file helps browsers recognize your app as installable.

OUTPUT :

manifest.json
maharashtra-forts-pwa > public > manifest.json > ...
{
"name": "Maharashtra Forts Explorer",
"short_name": "MH Forts",
"description": "Explore the magnificent forts of Maharashtra with offline access",
"start_url": "/",
"display": "standalone",
"background_color": "#ffffff",
"theme_color": "#1e3a8a",
"icons": [
{
"src": "pwa-192x192.png",
"sizes": "192x192",
"type": "image/png"
},
{
"src": "pwa-512x512.png",
"sizes": "512x512",
"type": "image/png"
}
]
}

localhost:5174
Raigad Fort
Pratapgad Fort
Sinhagad Fort
Shivneri Fort
Service workers
http://localhost:5174/
Network requests
Update
Unregister
Source dev-sw.js
Received 4/1/2025, 9:33:50 AM
Status #960 activated and is running
Stop
Clients http://localhost:5174/
Push
Test push message from DevTools.
Sync
test-tag-from-devtools
Periodic sync
Update Cycle
Version
Update Activity
Timeline
#960 Install
#960 Wait
#960 Activate

localhost:5174
महाराष्ट्राचे किल्ले
Explore the majestic forts of Maharashtra
Search forts by name, location or features...
Raigad Fort
Pratapgad Fort
Sinhagad Fort
Shivneri Fort
Raigad District
Satara District
Pune District
Junnar, Pune District

Conclusion:

By following these steps, we successfully converted our Vite React application into a Progressive Web App. This enables offline accessibility, caching for better performance, and an installable app experience on mobile and desktop devices. Implementing PWA features enhances user engagement and makes the app function seamlessly even in limited network conditions.

EXPERIMENT No. 8

Name : Arya Manoj Madhavi
Class : D15B
Roll No.: 29

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:

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

Things to note about Service Worker:
● A service worker is a programmable network proxy that lets you control how network requests from your page are handled.
● Service workers only run over HTTPS. Because service workers can intercept network requests and modify responses, "man-in-the-middle" attacks could be very bad.
● The service worker becomes idle when not in use and restarts when it's next needed. You cannot rely on a global state persisting between events. If there is information that you need to persist and reuse across restarts, you can use IndexedDB databases.

What can we do with Service Workers?
● You can dominate Network Traffic
You can manage all network traffic of the page and do any manipulations. For example, when the page requests a CSS file, you can send plain text as a response or when the page requests an HTML file, you can send a png file as a response. You can also send a true response too.
● You can Cache
You can cache any request/response pair with Service Worker and Cache API and you can access these offline content anytime.
● You can manage Push Notifications
You can manage push notifications with Service Worker and show any information message to the user.
● You can Continue Although Internet connection is broken, you can start any process with Background Sync of Service Worker.

What can’t we do with Service Workers?
● You can’t access the Window
You can’t access the window, therefore, You can’t manipulate DOM elements. But, you can communicate to the window through post Message and manage processes that you want.

● You can’t work it on 80 Port
Service Worker just can work on HTTPS protocol. But you can work on localhost during development. Service Worker Cycle

A service worker goes through three steps in its life cycle:
● Registration
● Installation
● Activation
Registration
To install a service worker, you need to register it in your main JavaScript code. Registration tells the browser where your service worker is located, and to start installing it in the background. Let's look at an example:
main.js

```
if ('serviceWorker' in navigator) {
navigator.serviceWorker.register('/service-worker.js')
.then(function(registration) {
console.log('Registration successful, scope is:', registration.scope);
})
.catch(function(error) {
console.log('Service worker registration failed, error:', error);
});
}
```

This code starts by checking for browser support by examining navigator.serviceWorker. The service worker is then registered with navigator.serviceWorker.register, which returns a promise that resolves when the service worker has been successfully registered. The scope of the service worker is then logged with registration.scope. If the service worker is already installed, navigator.serviceWorker.register returns the registration object of the currently active service worker.
The scope of the service worker determines which files the service worker controls, in other words, from which path the service worker will intercept requests. The default scope is the location of the service worker file, and extends to all directories below. So if service-worker.js is located in the root directory, the service worker will control requests from all files at this domain.

You can also set an arbitrary scope by passing in an additional parameter when registering. For example: main.js

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

In this case we are setting the scope of the service worker to /app/, which means the service worker will control requests from pages like /app/, /app/lower/ and /app/lower/lower, but not from pages like /app or /, which are higher.

If you want the service worker to control higher pages e.g. /app (without the trailing slash) you can indeed change the scope option, but you'll also need to set the Service-Worker-Allowed HTTP Header in your server config for the request serving the service worker script.

main.js

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

Installation

Once the browser registers a service worker, installation can be attempted. This occurs if the service worker is considered to be new by the browser, either because the site currently doesn't have a registered service worker, or because there is a byte difference between the new service worker and the previously installed one.

A service worker installation triggers an install event in the installing service worker. We can include an install event listener in the service worker to perform some task when the service worker installs. For instance, during the install, service workers can precache parts of a web app so that it loads instantly the next time a user opens it (see caching the application shell). So, after that first load, you're going to benefit from instant repeat loads and your time to interactivity is going to be even better in those cases. An example of an installation event listener looks like this:

service-worker.js

```
// Listen for install event, set callback
self.addEventListener('install', function(event) {
// Perform some task
});
```

Activation

Once a service worker has successfully installed, it transitions into the activation stage. If there are any open pages controlled by the previous service worker, the new service worker enters a waiting state. The new service worker only activates when there are no longer any pages loaded that are still using the old service worker. This ensures that only one version of the service worker is running at any given time.

When the new service worker activates, an activate event is triggered in the activating service worker. This event listener is a good place to clean up outdated caches (see the Offline Cookbook for an example).

service-worker.js

```
self.addEventListener('activate', function(event) {
// Perform some task
```

```
});
```

Once activated, the service worker controls all pages that load within its scope, and starts listening for events from those pages. However, pages in your app that were loaded before the service worker activation will not be under service worker control. The new service worker will only take over when you close and reopen your app, or if the service worker calls clients.claim(). Until then, requests from this page will not be intercepted by the new service worker. This is intentional as a way to ensure consistency in your site.

CODE :

sw.js

```
/**
 * Copyright 2018 Google Inc. All Rights Reserved.
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *     http://www.apache.org/licenses/LICENSE-2.0
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

// If the loader is already loaded, just stop.
if (!self.define) {
  let registry = {};

  // Used for `eval` and `importScripts` where we can't get script URL by other means.
  // In both cases, it's safe to use a global var because those functions are synchronous.
  let nextDefineUri;

  const singleRequire = (uri, parentUri) => {
    uri = new URL(uri + ".js", parentUri).href;
    return registry[uri] || (

        new Promise(resolve => {
          if ("document" in self) {
            const script = document.createElement("script");
            script.src = uri;
            script.onload = resolve;
            document.head.appendChild(script);
          } else {
            nextDefineUri = uri;
            importScripts(uri);
```

```
            resolve();
          }
        })

      .then(() => {
        let promise = registry[uri];
        if (!promise) {
          throw new Error(`Module ${uri} didn’t register its module`);
        }
        return promise;
      })
    );
  };

  self.define = (depsNames, factory) => {
    const uri = nextDefineUri || ("document" in self ? document.currentScript.src : "") ||
location.href;
    if (registry[uri]) {
      // Module is already loading or loaded.
      return;
    }
    let exports = {};
    const require = depUri => singleRequire(depUri, uri);
    const specialDeps = {
      module: { uri },
      exports,
      require
    };
    registry[uri] = Promise.all(depsNames.map(
      depName => specialDeps[depName] || require(depName)
    )).then(deps => {
      factory(...deps);
      return exports;
    });
  };
}
define(['./workbox-d9a5ed57'], (function (workbox) { 'use strict';

  self.skipWaiting();
  workbox.clientsClaim();

  /**
   * The precacheAndRoute() method efficiently caches and responds to
   * requests for URLs in the manifest.
```

```
 * See https://goo.gl/S9QRab
 */
workbox.precacheAndRoute([{
  "url": "registerSW.js",
  "revision": "3ca0b8505b4bec776b69afdba2768812"
}, {
  "url": "index.html",
  "revision": "0.7e7jl0981k"
}], {});
workbox.cleanupOutdatedCaches();
workbox.registerRoute(new
workbox.NavigationRoute(workbox.createHandlerBoundToURL("index.html"), {
  allowlist: [/^\/$/]
}));
workbox.registerRoute(/\.(?:png|jpg|jpeg|svg)$/, new workbox.CacheFirst({
  "cacheName": "images-cache",
  plugins: [new workbox.ExpirationPlugin({
    maxEntries: 50,
    maxAgeSeconds: 2592000
  })]
}), 'GET');

}));
```

OUTPUT :

localhost:5175
महाराष्ट्राचे किल्ले
Explore the majestic forts of Maharashtra
Elements
Console
Sources
Network
Memory
Application
Performance
Privacy and security
Lighthouse
Recorder
Components
Profiler
Application
Manifest
Service workers
Storage
Storage
Local storage
Session storage
Extension storage
IndexedDB
Cookies
Private state tokens
Interest groups
Shared storage
Cache storage
Storage buckets
Background services
Back/forward cache
Background fetch
Background sync
Bounce tracking miti...
Notifications
Storage
http://localhost:5175/
Usage
1.0 MB used out of 227,883 MB storage quota
Learn more
1.0 MB
960 kB Service workers
42.0 kB Cache storage
3.5 kB IndexedDB
1.0 MB Total
Simulate custom storage quota
Clear site data
including third-party cookies
Application
Unregister service workers
AI assistance
Console
What's new
Issues

maharashtra-forts-pwa
dev-dist
registerSW.js
sw.js
workbox-d9a5ed5...
node_modules
public
images
favicon.ico
manifest.json
pwa-192x192.png
pwa-512x512.png
vite.svg

localhost:5174
Install app
Maharashtra Forts Explorer
localhost:5174
Install
Cancel
Explore the majestic forts of Maharashtra
Search forts by name, location or features...
Raigad Fort
Pratapgad Fort
Sinhagad Fort
Shivneri Fort
Raigad District
Satara District
Pune District
Junnar, Pune District

CONCLUSION :

Service workers are essential for PWAs, enabling offline access, caching, and efficient network request handling. In this implementation, the service worker is registered, installed, and activated using **Workbox**, ensuring seamless updates and improved performance. By precaching assets and using runtime caching strategies, it enhances page load speed, reduces server load, and allows access to content even without an internet connection. This makes the e-commerce PWA more reliable, responsive, and user-friendly.

EXPERIMENT No. 9

Name : Arya Manoj Madhavi
Class : D15B
Roll No.: 29

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

Theory:
Service Worker

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

Things to note about Service Worker:

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

Fetch Event

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

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

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

Sync Event

Background Sync is a Web API that is used to delay a process until the Internet connection is stable. We can

adapt this definition to the real world; there is an e-mail client application that works on the browser and we want to send an email with this tool. Internet connection is broken while we are writing e-mail content and we didn't realize it. When completing the writing, we click the send button.

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

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

1. When we click the "send" button, email content will be saved to IndexedDB.
2. Background Sync registration.
3. **If the Internet connection is available**, all email content will be read and sent to Mail Server. **If the Internet connection is unavailable**, the service worker waits until the connection is available even though the window is closed. When it is available, email content will be sent to Mail Server.

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

Push Event

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

with received data.
We can check in the following example.
“Notification.requestPermission();” is the necessary line to show notification to the user. If you don’t want to show any notification, you don’t need this line.

In the following code block is in sw.js file. You can handle push notifications with this event. In this example, I kept it simple. We send an object that has “method” and “message” properties. If the method value is “pushMessage”, we open the information notification with the “message” property.

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

Code:
sw.js

```
/**
 * Copyright 2018 Google Inc. All Rights Reserved.
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *     http://www.apache.org/licenses/LICENSE-2.0
 * Unless required by applicable law or agreed to in writing, software
 * distributed under the License is distributed on an "AS IS" BASIS,
 * WITHOUT WARRANTIES OR CONDITIONS OF ANY KIND, either express or implied.
 * See the License for the specific language governing permissions and
 * limitations under the License.
 */

// If the loader is already loaded, just stop.
```

```
if (!self.define) {
  let registry = {};

  // Used for `eval` and `importScripts` where we can't get script URL by other means.
  // In both cases, it's safe to use a global var because those functions are synchronous.
  let nextDefineUri;

  const singleRequire = (uri, parentUri) => {
    uri = new URL(uri + ".js", parentUri).href;
    return registry[uri] || (

        new Promise(resolve => {
          if ("document" in self) {
            const script = document.createElement("script");
            script.src = uri;
            script.onload = resolve;
            document.head.appendChild(script);
          } else {
            nextDefineUri = uri;
            importScripts(uri);
            resolve();
          }
        })

      .then(() => {
        let promise = registry[uri];
        if (!promise) {
          throw new Error(`Module ${uri} didn’t register its module`);
        }
        return promise;
      })
    );
  };

  self.define = (depsNames, factory) => {
    const uri = nextDefineUri || ("document" in self ? document.currentScript.src : "") || location.href;
    if (registry[uri]) {
      // Module is already loading or loaded.
      return;
    }
    let exports = {};
    const require = depUri => singleRequire(depUri, uri);
    const specialDeps = {
      module: { uri },
      exports,
      require
    };
    registry[uri] = Promise.all(depsNames.map(
      depName => specialDeps[depName] || require(depName)
    )).then(deps => {
```

```
      factory(...deps);
      return exports;
    });
  };
}
define(['./workbox-d9a5ed57'], (function (workbox) { 'use strict';

  self.skipWaiting();
  workbox.clientsClaim();

  /**
   * The precacheAndRoute() method efficiently caches and responds to
   * requests for URLs in the manifest.
   * See https://goo.gl/S9QRab
   */
  workbox.precacheAndRoute([{
    "url": "registerSW.js",
    "revision": "3ca0b8505b4bec776b69afdba2768812"
  }, {
    "url": "index.html",
    "revision": "0.7e7jl0981k"
  }], {});
  workbox.cleanupOutdatedCaches();
  workbox.registerRoute(new
workbox.NavigationRoute(workbox.createHandlerBoundToURL("index.html"), {
    allowlist: [/^\/$/]
  }));
  workbox.registerRoute(/\.(?:png|jpg|jpeg|svg)$/, new workbox.CacheFirst({
    "cacheName": "images-cache",
    plugins: [new workbox.ExpirationPlugin({
      maxEntries: 50,
      maxAgeSeconds: 2592000
    })]
  }), 'GET');

}));
```

Output:

Fetch event

Sync event and Push event

OFFLINE ACCESS

Without Network

BY Download

in System

Conclusion

Service workers play a vital role in making Progressive Web Apps (PWAs) more efficient, reliable, and user-friendly. They enable background processes such as caching (fetch event), delayed tasks for better connectivity (sync event), and push notifications (push event). By implementing these service worker events, the e-commerce PWA ensures better performance,

offline accessibility, seamless user experience, and reduced server load. Using Workbox simplifies service worker management, making it easier to handle caching strategies, push notifications, and background synchronization, ultimately enhancing the reliability and responsiveness of the web application.

EXPERIMENT No. 10

Name : Arya Manoj Madhavi
Class : D15B
Roll No.: 29

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

Theory:

GitHub Pages
Public web pages are freely hosted and easily published. Public webpages hosted directly from your GitHub repository. Just edit, push, and your changes are live.

GitHub Pages provides the following key features:
1. Blogging with Jekyll
2. Custom URL
3. Automatic Page Generator

Reasons for favoring this over Firebase:
1. Free to use
2. Right out of github
3. Quick to set up

GitHub Pages is used by Lyft, CircleCI, and HubSpot.
GitHub Pages is listed in 775 company stacks and 4401 developer stacks.

Pros
1. Very familiar interface if you are already using GitHub for your projects.
2. Easy to set up. Just push your static website to the gh-pages branch and your website is ready.
3. Supports Jekyll out of the box.
4. Supports custom domains. Just add a file called CNAME to the root of your site, add an A record in the site's DNS configuration, and you are done.

Cons
1. The code of your website will be public, unless you pay for a private repository.
2. Currently, there is no support for HTTPS for custom domains. It's probably coming soon though.
3. Although Jekyll is supported, plug-in support is rather spotty.

Firebase
The Realtime App Platform. Firebase is a cloud service designed to power real-time,

collaborative applications. Simply add the Firebase library to your application to gain access to a shared data structure; any changes you make to that data are automatically synchronized with the
Firebase cloud and with other clients within milliseconds.

Some of the features offered by Firebase are:
1. Add the Firebase library to your app and get access to a shared data structure. Any changes made to that data are automatically synchronized with the Firebase cloud and with other clients within milliseconds.
2. Firebase apps can be written entirely with client-side code, update in real-time out-of-the-box, interoperate well with existing services, scale automatically, and provide strong data security.
3. Data Accessibility- Data is stored as JSON in Firebase. Every piece of data has its own URL which can be used in Firebase's client libraries and as a REST endpoint. These URLs can also be entered into a browser to view the data and watch it update in Real-time.

Reasons for favoring over GitHub Pages:
1. Realtime backend made easy
2. Fast and responsive
Instacart, 9GAG, and Twitch are some of the popular companies that use Firebase
Firebase has a broader approval, being mentioned in 1215 company stacks & 4651 developers Stacks

Pros
1. Hosted by Google. Enough said.
2. Authentication, Cloud Messaging, and a whole lot of other handy services will be available to you.
3. A real-time database will be available to you, which can store 1 GB of data.
4. You'll also have access to a blob store, which can store another 1 GB of data.
5. Support for HTTPS. A free certificate will be provisioned for your custom domain within 24 hours.

Cons
1. Only 10 GB of data transfer is allowed per month. But this is not really a big problem, if you use a CDN or AMP.
2. Command-line interface only.
3. No in-built support for any static site generator.

Link to our GitHub repository: https://github.com/aryamadhavi03/PWA

Github Screenshot:

Conclusion:

The deployment of the E-commerce Progressive Web App (PWA) to GitHub Pages has been successfully studied and implemented. Through this process, we understood the essential steps involved in converting a React-based E-commerce site into a PWA, enabling offline access, faster loading, and app-like behavior. Furthermore, we explored GitHub Pages as a free and effective static hosting platform, learned how to configure the repository and build files for deployment, and resolved challenges such as routing issues and service worker configuration. This hands-on experience has strengthened our understanding of both PWA architecture and modern deployment techniques.

EXPERIMENT No. 11

Name : Arya Manoj Madhavi
Class : D15B
Roll No.: 29

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

Theory :
Reference : https://www.semrush.com/blog/google-lighthouse/

Google Lighthouse :
Google Lighthouse is a tool that lets you audit your web application based on a number of parameters including (but not limited to) performance, based on a number of metrics, mobile compatibility, Progressive Web App (PWA) implementations, etc. All you have to do is run it on a page or pass it a URL, sit back for a couple of minutes and get a very elaborate report, not much short of one that a professional auditor would have compiled in about a week.
The best part is that you have to set up almost nothing to get started. Let's begin by looking at some of the top features and audit criteria used by Lighthouse.
Key Features and Audit Metrics
Google Lighthouse has the option of running the Audit for Desktop as well as mobile version of your page(s).

The top metrics that will be measured in the Audit are:
1. Performance: This score is an aggregation of how the page fared in aspects such as (but not limited to) loading speed, time taken for loading for basic frame(s), displaying meaningful content to the user, etc. To a layman, this score is indicative of how decently the site performs, with a score of 100 meaning that you figure in the 98th percentile, 50 meaning that you figure in the 75th percentile and so on.
2. PWA Score (Mobile): Thanks to the rise of Service Workers, app manifests, etc., a lot of modern web applications are moving towards the PWA paradigm, where the objective is to make the application behave as close as possible to native mobile applications. Scoring points are based on the Baseline PWA checklist laid down by Google which includes Service Worker implementation(s), viewport handling, offline functionality, performance in script-disabled environments, etc.
3. Accessibility: As you might have guessed, this metric is a measure of how accessible your website is, across a plethora of accessibility features that can be implemented in your page (such as the 'aria-' attributes like aria-required, audio captions, button names, etc.). Unlike the other metrics though, Accessibility metrics score on a pass/fail basis i.e. if all possible elements of the page are not screen-reader friendly (HTML5 introduced features that would make pages easy to interpret for screen readers used by visually challenged people like tag names, tags such as <section>, <article>, etc.), you get a 0 on that score. The aggregate of these scores is your Accessibility metric score.

4. Best Practices: As any developer would know, there are a number of practices that have been deemed ‘best’ based on empirical data. This metric is an aggregation of many such points, including but not limited to:Use of HTTPS
Avoiding the use of deprecated code elements like tags, directives, libraries, etc.
Password input with paste-into disabled
Geo-Location and cookie usage alerts on load, etc.

Changes made to the code :
For theme color add a meta tag in index.html-
<meta name="theme-color" content="#4285f4">

For a maskable icon add "purpose": "any maskable" to the icons in manifest.json file
For apple touch icon add the following meta tag in index.html-

<link rel="apple-touch-icon" href="">

Changes in manifest.json

```
{
"name": "flower shop website",
"short_name": "flowers",
"start_url": "index.html",
"scope": "./",
"theme_color": "#ffd31d",
"background_color": "#333",
"display": "standalone",
"icons": [
{
"src": "icon-1.png",
"sizes": "192x192",
"type": "image/png"
“purpose”:"any maskable"
},
{
"src": "icon-2.png",
"sizes": "512x512",
"type": "image/png"
“purpose”:"any maskable"
}
]
}
```

Conclusion:

Thus, we successfully used the Google Lighthouse PWA Analysis Tool to evaluate the performance and compliance of our Progressive Web App. The tool provided detailed insights into various parameters such as performance, accessibility, best practices, and PWA capabilities. It helped us identify strengths and areas for improvement. Overall, it proved to be an effective and user-friendly auditing solution for testing PWA functionality.

Name: Anya Manoj Madhavi
Class: D15B
Roll no: 29

05 A+

MPL Assignment 1

Q.1.a) Explain the key features and advantages of using Flutter for mobile app development.

→ Key features of Flutter:

① Single Codebase - Write one code for both Android and iOS
② Fast Performance - Uses Dart Language and high performance rendering engine.
③ Hot Reload - See changes instantly without restarting the app
④ Rich UI Components - Comes with customizable widgets for smooth UI design.
⑤ Native like Experience - Provides high quality animations and fast execution.
⑥ Cross - Platform Support - Can be used for mobile, web and desktop apps.
⑦ Open - Source - Free to use and has a strong developer community.

Advantages of using Flutter:

① Saves Time and Effort - Single codebase for multiple platforms.
② Cost Effective - Reduces development cost and time.
③ Attractive UI - Provides beautiful and customizable widgets.
④ Good Performance - Uses Dart and Skia for fast and smooth rendering.
⑤ Easy Integration - Supports third party plugins and native code integration.
⑥ High Speed Development - Hot reload features speeds up coding.

Q1. b) Discuss how the Flutter framework differs from traditional approaches and why it has gained popularity in the developer community

→ How Flutter differs from Traditional Approaches:

① Single Codebase - Traditional methods need separate code for Android (Java / Kotlin) and iOS (Swift / Objective-C), but Flutter uses one code for both.

② Hot Reload - Traditional apps require full restart after changes, but Flutter updates instantly.

③ UI rendering - Flutter has its own rendering engine (Skia) for faster performance.

④ Performance - Flutter compiles directly to native machine code, making it faster than frameworks that use a bridge.

⑤ Customization - Traditional UI design depends on platform-specific components but Flutter provides fully customizable widgets.

Why Flutter is popular among developers:

① Fast Development - Hot reload and single codebase save time.

② Cross-Platform Support - Works on mobile web and desktop.

③ Beautiful UI - Rich, customizable widgets for modern designs.

④ High performance - runs smoothly without a bridge like React Native.

Q.2.a) Describe the concept of widget tree in Flutter. Explain how widget composition is used to build complex user interfaces.

→ Concept of widget tree in Flutter :

① In flutter, everything is a widget, widgets are arranged in tree structure, called the widget tree. This tree represents the UI of app, where parent widgets contains child widgets.

② Eg : A scaffold widget can have a Column widget, which contains Text and Button widgets. Changes in widgets update the tree dynamically.

③ Widget Composition for Complex UI :
Flutter uses small, reusable widgets to build complex UI. Instead of creating a single large UI block, developers combine multiple small widgets.

④ Eg : 1) A ListView can contain multiple Card widgets
2) A Column can hold Text, Image and Buttons.

This modular approach makes the UI flexiable, readable and easy to manage.

Q.2.b) Provide examples of commonly used widgets and their roles in creating a widget tree.

→ Commonly used widgets and their roles in a widget tree :

① Scaffold : Provides the basic layout structure.
② App Bar : Displays the top navigation bar with a title.

③ Text : Displays simple text on the screen.
④ Image : Shows images from assets or URL's.
⑤ Container - Used for styling.
⑥ ListView - Displays scrollable lists.

Example widget Tree :

This tree structure helps in organizing and managing the UI efficiently.

Q.3.a) Discuss the importance of state management in Flutter applications.

→ Importance of State Management in Flutter Applications

Statem management is important because it controls how the app stores, updates and displays data when the user interacts with it.

Why State Management is Needed?

1. Keeps UI updated
2. Improves performance
3. Manages Complex Data
4. Ensures Smooth User Experience.

Types of State in Flutter:

1. Local State - Managed within a single widget using Stateful Widget.

2. Global State - Shared across multiple screens using Provider, Riverpod, Bloc or Redux.

Without proper state management, the app may behave unpredictably or show outdated data.

Q.3.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.

Approach	How it works	When to use
setState	Updates UI by calling setState() in a StatefulWidget	Best for small apps or managing state within a single widget. eg: Toggling a button color.
Provider	Uses InheritedWidget to share state across widgets efficiently.	Suitable for medium sized apps where data needs to be shared between multiple widgets. eg: managing user authentication.
Riverpod	An improved version of Provider with better performance and simpler syntax	Best for large apps that need complex state management with dependency injection. eg: Handling API data and app wide themes.

Q.4.a) Explain the process of integrating Firebase with a Flutter application. Discuss the benefits of using Firebase as a backend solution.

→ Process of Integrating firebase with Flutter Application:

① Create a Firebase Project - Go to Firebase Console, create a new project.
② Add Firebase to Flutter App - Register the app (Android/ios) and download the google services.json (Android) or Google Service-Info.plist (ios).
③ Install Firebase Packages - Add dependencies like 'firebase-core' and 'firebase_auth' in 'pubspec.yaml'.
④ Initialize Firebase - Import Firebase in 'main.dart' and call 'Firebase.initializeApp()'.
⑤ Use Firebase Services - Implement authentication, database or cloud functions as needed.

Benefits of using firebase as a Backend Solution:
① Real-time Database
② Authentication
③ Cloud Firestore
④ Hosting and Storage
⑤ Scalability
⑥ Push Notifications.

Q.4. b) Highlight the firebase services commonly used in Flutter development and provide a brief overview of how data synchronization is achieved.

→ Common Firebase Services used in Flutter Development.

1. Firebase Authentication
2. Cloud Firestore
3. Firebase Realtime Database
4. Firebase Cloud Storage
5. Firebase Cloud Messaging (FCM)
6. Firebase Hosting
7. Firebase Analytics

How data Synchronization is Achieved:

1. Real time updates - Firestore and Realtime Database sync data across devices instantly.
2. Listeners & Streams - widgets listen for changing and updates the UI automatically.
3. Offline Support - Firebase caches data, allowing apps to work offline and sync when online.

This ensures fast, smooth and automatic data updates in Flutter apps.

Name: Anya Manoj Madhavi
Class: D15B
Roll no: 29

05/05

MPL Assignment 2

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

→ A Progressive Web App (PWA) is a type of web application that works like a mobile app but runs in a browser. It can be installed on a device, works offline and provides a fast and smooth user experience.

Significance of PWA in Modern Web Development:
1. Cross-Platform Compatibility
2. Offline Support
3. Fast Performance
4. No App Store Required
5. Lower Development Cost.

Key differences Between PWA & Traditional Mobile Apps:

Features	PWA	Traditional Mobile App
Installation	Direct from browser	Download from App Store
Internet Required	Works offline with caching	Usually requires internet.
Performance	Fast with service workers	Faster but needs installation.

Updates	Automatic. no app store approval	Manual updates needed
Development Cost	Lower (one codebase for all)	Higher (separate apps for each platform)

PWA's combine the best of web and mobile apps, making them efficient and user friendly.

Q.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.

→ Definition of Responsive web Design :
Responsive Web Design (RWD) is technique that makes web pages adjust automatically to different screen sizes and devices. It ensures a good user experience on mobile, tablets and desktops without needing separate versions of website.

Importance of Responsive Design in PWA's :

① Better User Experience - PWA's work smoothly on any device.
② Faster Load Time - Optimized design improves speed
③ SEO Benefits - Google ranks responsive sites higher.

④ Cost Effective - No need to build multiple versions for different screens.

Comparison of Web Design Approaches :

Approach	How it works	Pros	~~Cons~~ Cons
Responsive	Uses flexible grids and CSS media queries to adjust layout.	Works on all devices.	Can be complex to design
Fluid	Uses percent-based widths instead of fixed pixels, so elements resize smoothly.	Works well on different screen sizes, easy to implement.	Less control over layout on large screens.
Adaptive	Uses fixed layouts that change at specific breakpoints	Optimized for known screen sizes	More effort required to design for each screen size.

Key Differences :

- Responsive adapts dynamically to all screens.
- Fluid resizes smoothly but may not be fully optimized
- Adaptive loads different layouts based on device types.

Responsive design is best for PWA's because it ensures a seamless experience on all devices.

Q.3. Describe the lifecycle of service workers, including registration, installation and activation phases.

→ Lifecycle of Service workers:
A Service worker is a script that runs in the background and helps a web app work offline, load faster, and send push notifications. Its lifecycle has three main phases.

① Registration Phase:
• The browser registers the Service worker using Javascript

Code Example:

```
if ('serviceworker' in navigator) {
  navigator.serviceworker.register('/sw.js');
  .then(() => console.log('Service worker
  Registered'))
  .cath(error => console.log('Registration Failed:',
  error));
```

• This tells the browser to install and activate the Service worker.

② Installation Phase
• The service worker downloads necessary files (HTML CSS, JS) and stores them in cache.

- If successful, it moves to the activation phase.

Code Example:

```
self.addEventListener('install', event => {
    event.waitUntil(
        caches.open('app-cache').then(cache => {
            return cache.addAll(['/', '/index.html',
            'styles.css']);
        })
    );
});
```

- This ensures the app loads even without the internet.

③ Activation Phase:

- The old service worker is replaced with the new one.
- Unused cache files from the previous version are deleted.

Final Step: Fetch & Sync

Once activated, the service worker intercepts network requests, serves cached files and syncs data when the Internet is available.

This lifecycle makes PWA's faster, more reliable and capable of working offline.

Q.4. Explain the use of Indexed DB in the System Service worker for data storage.

→ Use of Indexed DB in Service worker for Data Storage:
Indexed DB is a browser database that stores large amounts of structured data like JSON objects. It helps PWA's work offline by saving and retrieving data efficiently.

Why use Indexed DB in Service workers?

① Office Support - Stores data when offline and syncs it later.
② Efficient Storage - Saves structured data like user settings, cart items or form inputs.
③ Faster Access - Retrieves data quickly without needing a network request.
④ Persistent Data - Data remains saved even after the browser is closed.

How service workers use Indexed DB?

Opening the Database

```
let db;
let request = indexedDB.open('MyDatabase', 1);
request.onsuccess = function (event) {
    db = event.target.result;
};
```

Creating a Store & Adding Data

```
request.onupgradeneeded = function (event) {
let db = event.target.result;
let store = db.createObjectStore('Users', { keyPath: 'id' });
store.add({ id: 1, name: 'John Doe', age: 25 });
};
```

Fetching Data in Service Worker

```
let transaction = db.transaction('Users', 'readonly');
let store = transaction.objectStore('Users');
let getUser = store.get(1);

getUser.onsuccess = function() {
    console.log (getUser.result);
};
```