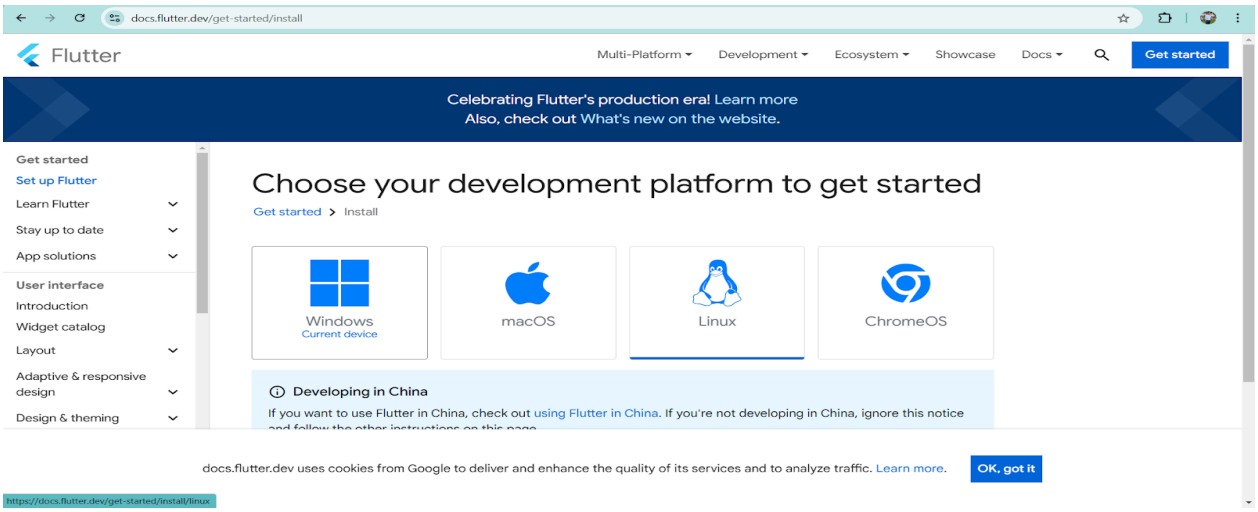
Name:Ajinkya Sunil Patil
D15B/42
Lab-mpl

EXP 1: Installation and Configuration of Flutter Environment.

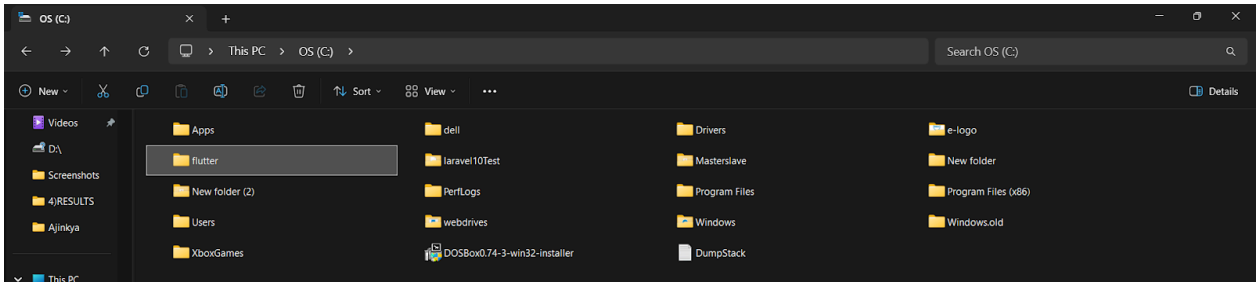
Install the Flutter SDK

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.



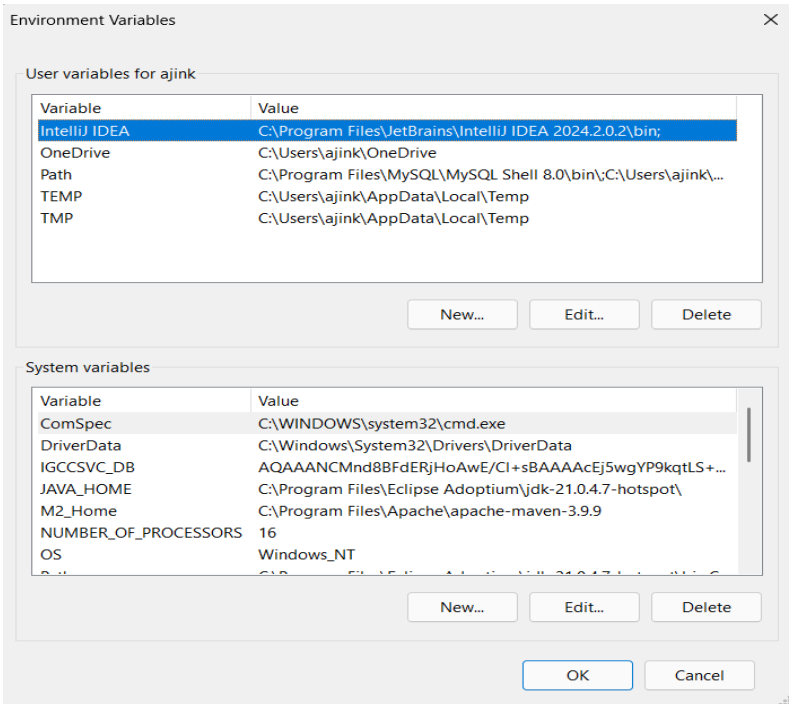
Step 2: Next, to download the latest Flutter SDK, click on the Windows icon. Here, you will find the download link for SDK.

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.

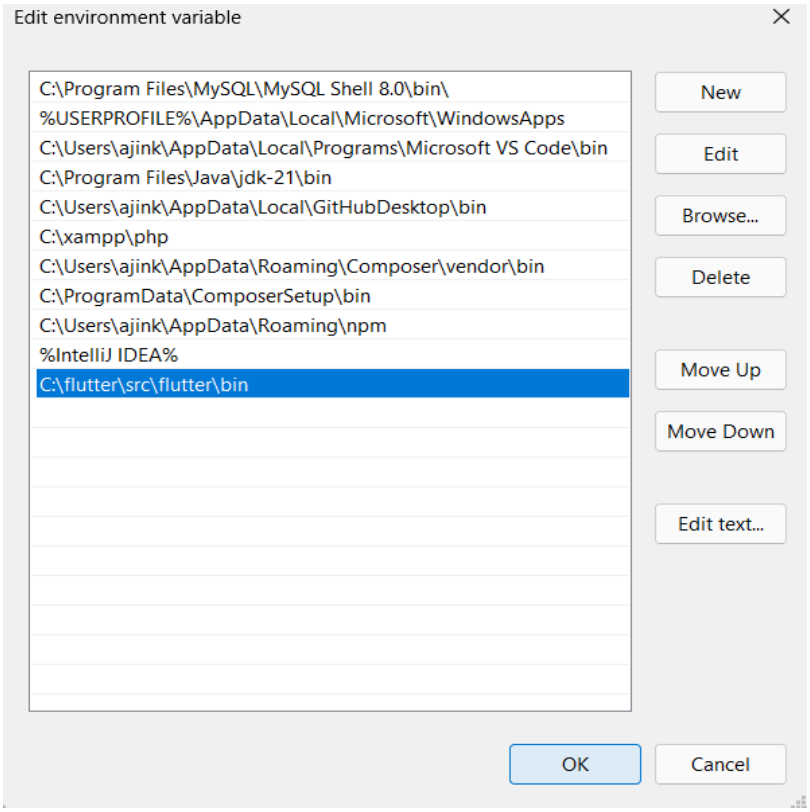


Step 4: To run the Flutter command in the regular windows console, you need to update the system path to include the flutter bin directory. The following steps are required to do this:

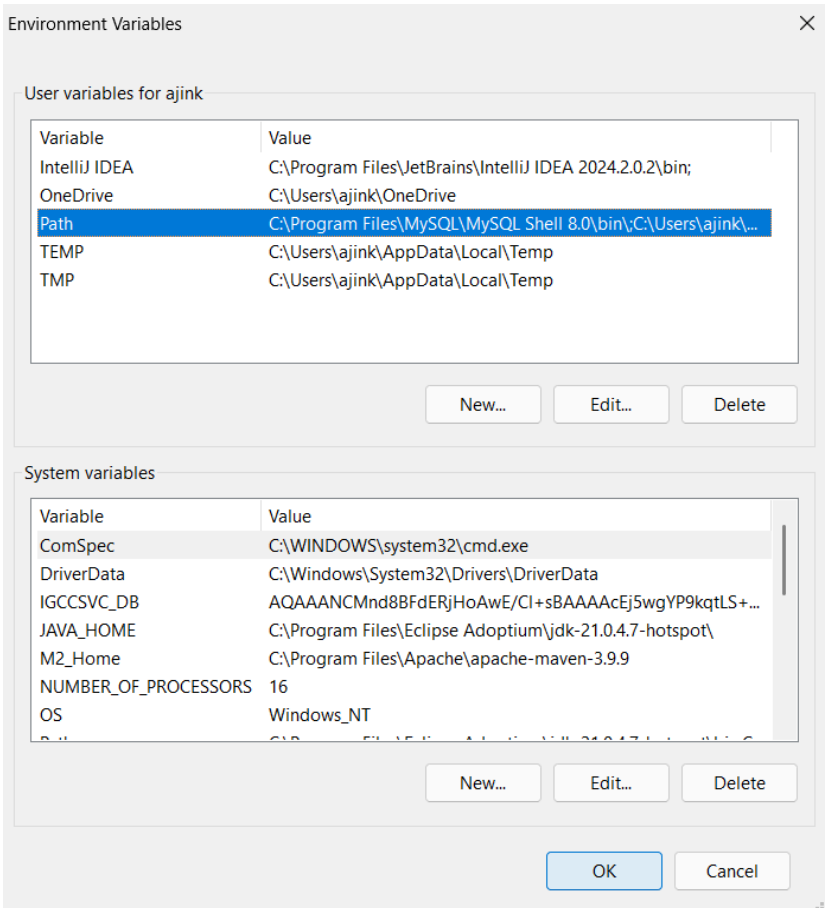
Step 4.1: Go to MyComputer properties -> advanced tab -> environment variables. You will get the following screen.



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

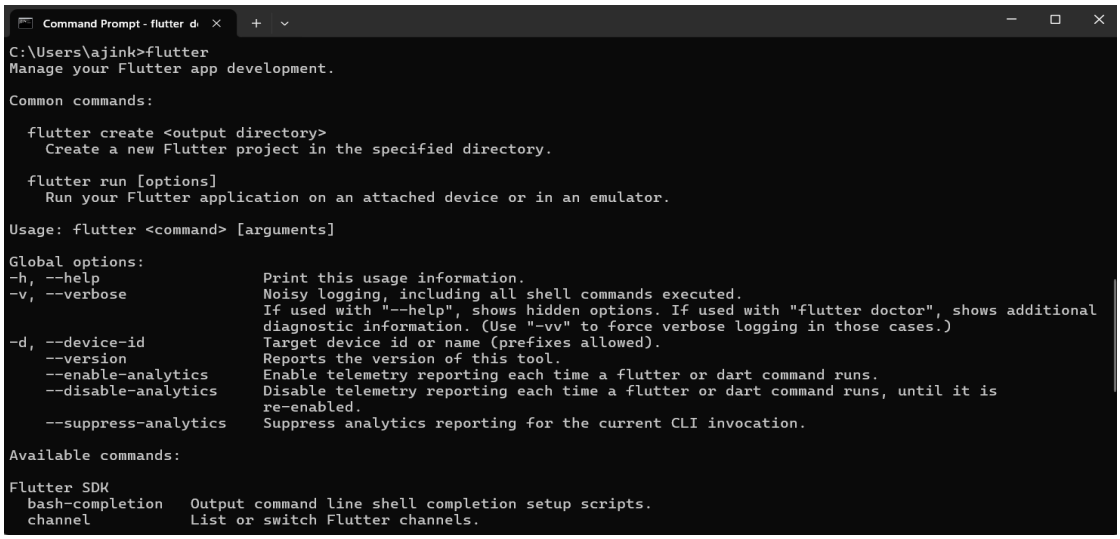


Step 4.3: In the above window, click on New->write path of Flutter bin folder in variable value - > ok -> ok -> ok.



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

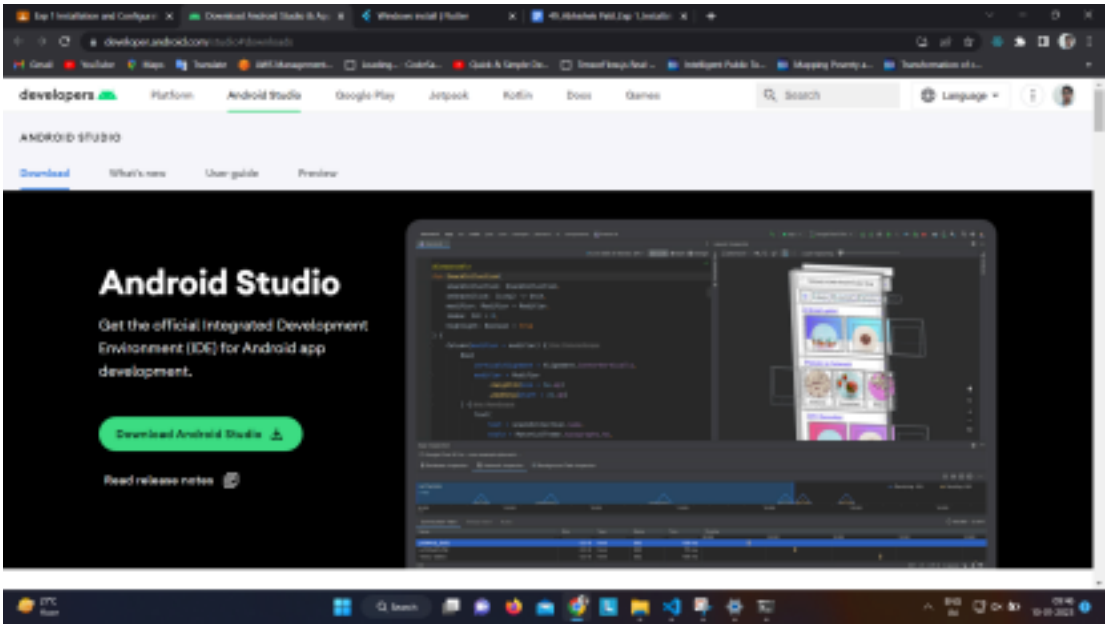


Step 6: When you run the above command, it will analyze the system and show its report, as shown in the below image. Here, you will find the details of all missing tools, which required to run Flutter as well as the development tools that are available but not connected with the device.

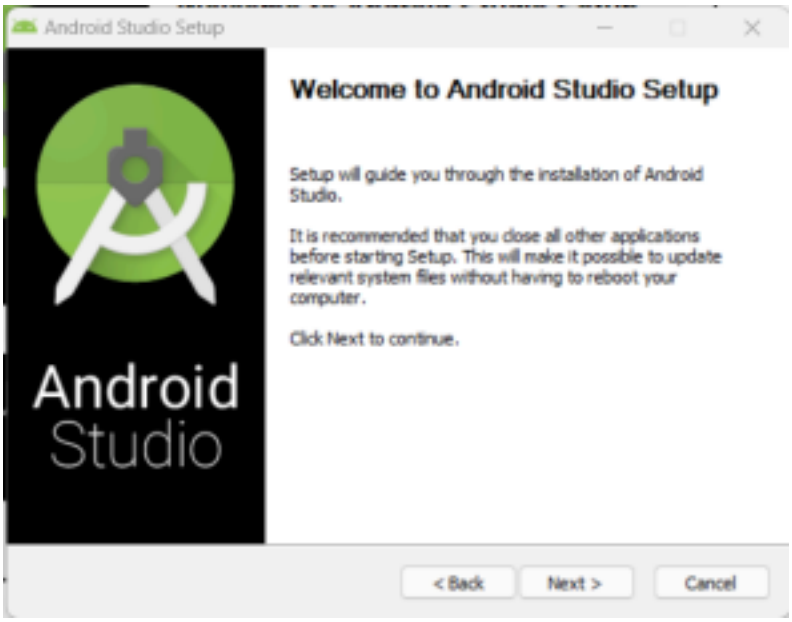
```
Command Prompt - flutter d

C:\Users\ajink>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
[X] Visual Studio - develop Windows apps
    X Visual Studio not installed; this is necessary to develop Windows apps.
      Download at https://visualstudio.microsoft.com/downloads/.
      Please install the "Desktop development with C++" workload, including all of its default components
[✓] Android Studio (version 2024.2)
[✓] IntelliJ IDEA Ultimate Edition (version 2024.2)
[✓] VS Code (version 1.96.4)
[✓] Connected device (4 available)
[!] Network resources
    X A cryptographic error occurred while checking "https://storage.googleapis.com/": Connection terminated during
      handshake
      You may be experiencing a man-in-the-middle attack, your network may be compromised, or you may have malware
      installed on your computer.
    X A cryptographic error occurred while checking "https://cocoapods.org/": Connection terminated during handshake
      You may be experiencing a man-in-the-middle attack, your network may be compromised, or you may have malware
      installed on your computer.
    X An HTTP error occurred while checking "https://github.com/": Connection closed before full header was received

! Doctor found issues in 2 categories.
```

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.

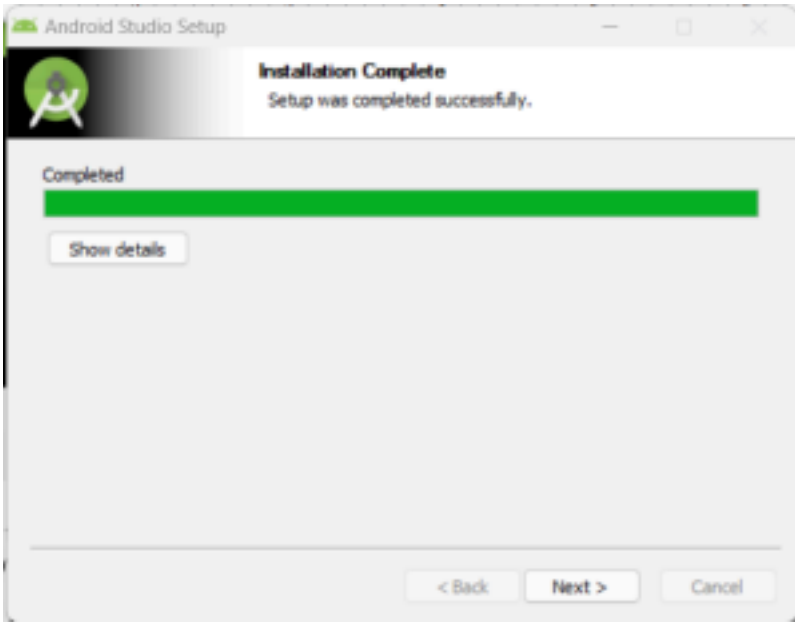
Step 7.1: Download the latest Android Studio executable or zip file from the official site.



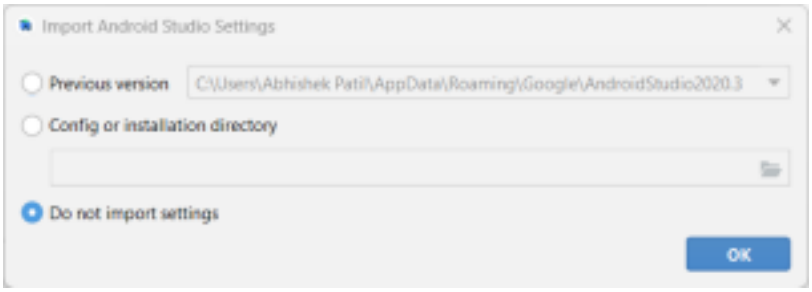
Step 7.2: When the download is complete, open the .exe file and run it. You will get the following dialog box.



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



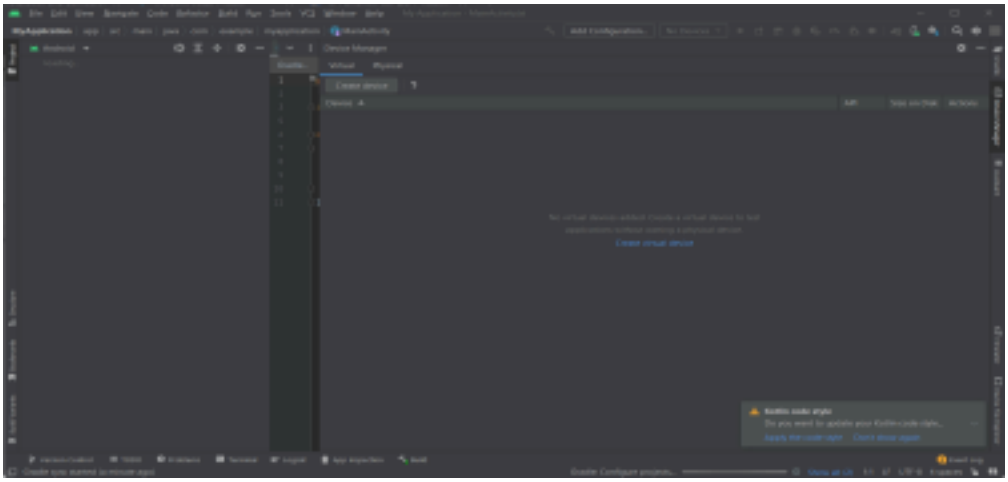
choose the 'Don't import Settings option' and click OK. It will start the Android Studio.

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

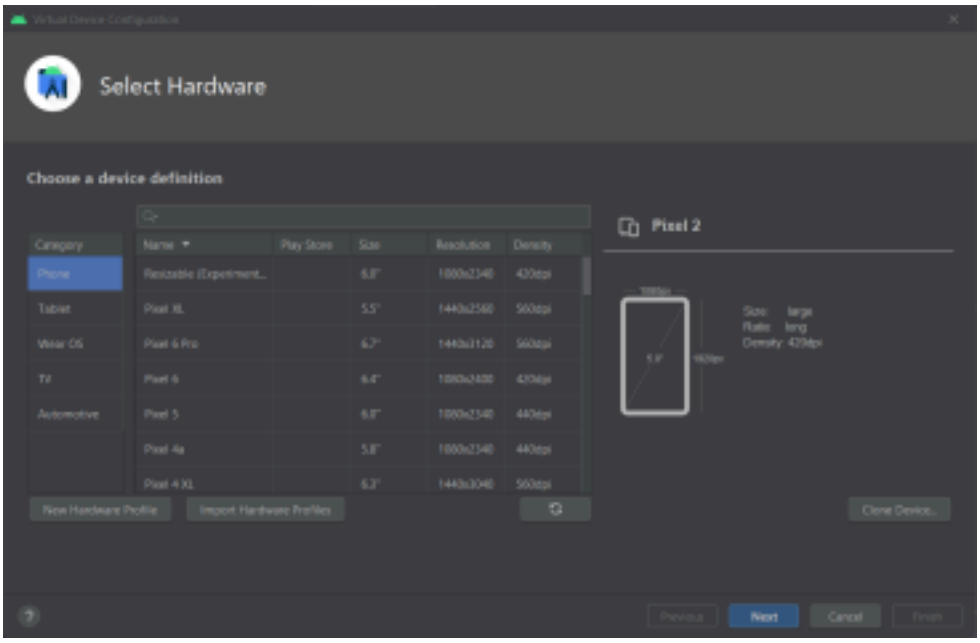
```
C:\Users\ajink>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 3.27.3, on Microsoft Windows [Version 10.0.26100.2894], locale en-IN)
[✓] Windows Version (Installed version of Windows is version 10 or higher)
[✓] Android toolchain - develop for Android devices (Android SDK version 35.0.1)
[✓] Chrome - develop for the web
[X] Visual Studio - develop Windows apps
    X Visual Studio not installed; this is necessary to develop Windows apps.
      Download at https://visualstudio.microsoft.com/downloads/.
      Please install the "Desktop development with C++" workload, including all of its default components
[✓] Android Studio (version 2024.2)
[✓] IntelliJ IDEA Ultimate Edition (version 2024.2)
[✓] VS Code (version 1.96.4)
[✓] Connected device (4 available)
[✓] Network resources

! Doctor found issues in 1 category.
```

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

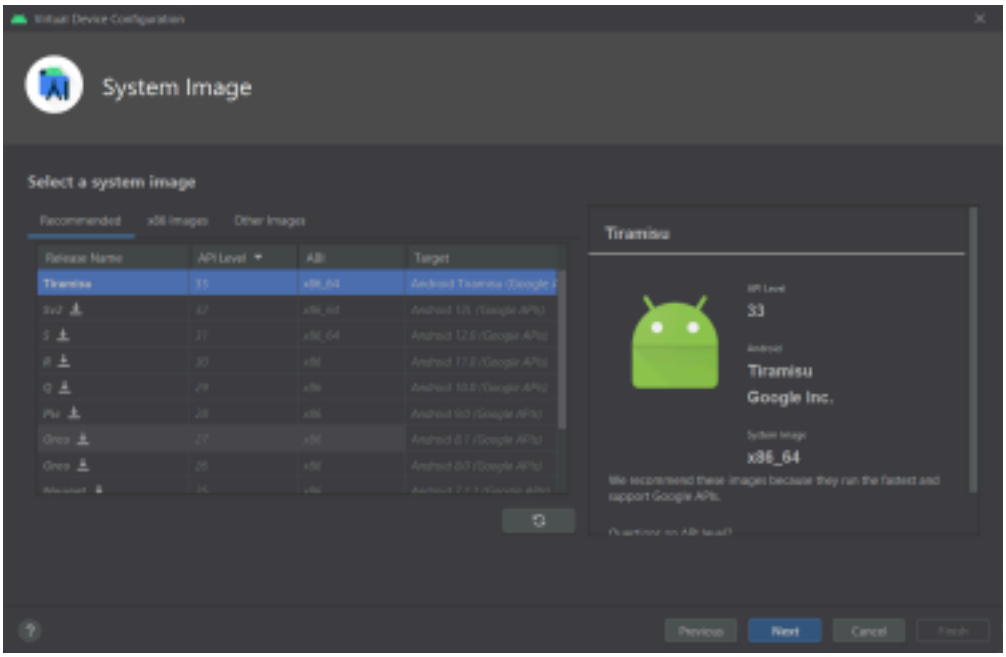
Step 8.1: To set an Android emulator, go to Android Studio > Tools > Android > AVD Manager and select Create Virtual Device. Or, go to Help->Find Action->Type Emulator in the search box. You will get the following screen.



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

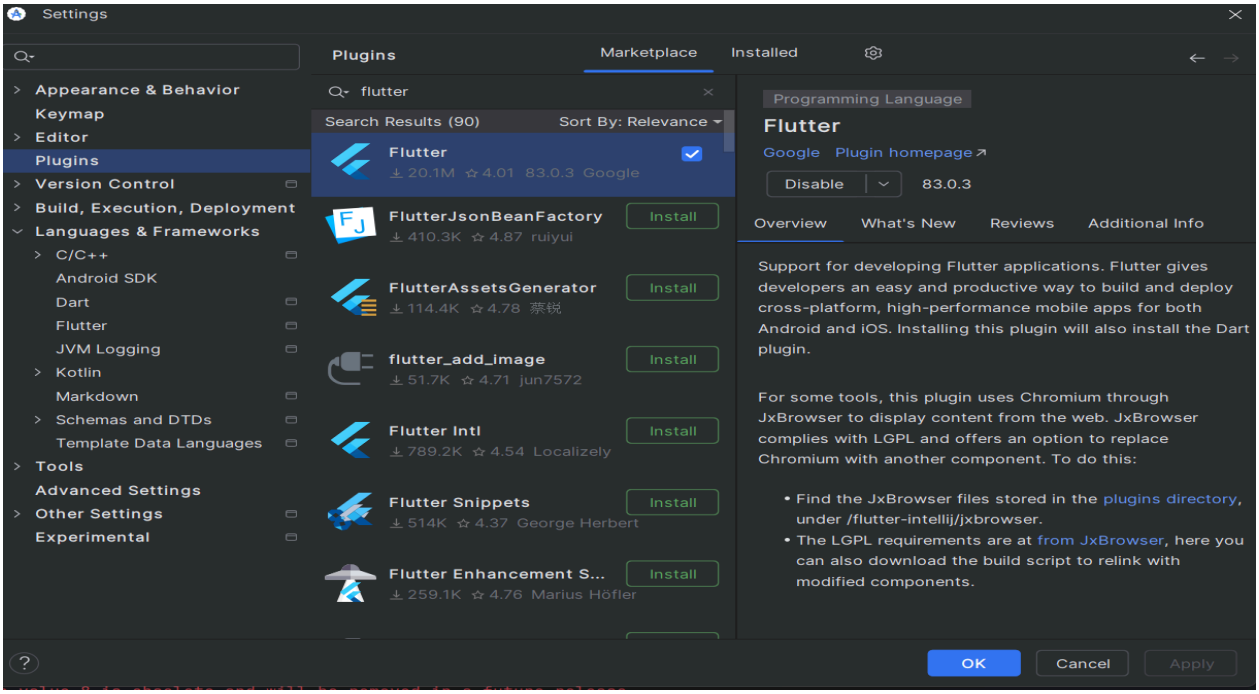


Step 8.3: Select the system image for the latest Android version and click on Next.
Step 8.4: Now, verify the all AVD configuration. If it is correct, click on Finish. The following screen appears.



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 Flutter and Dart plugin for building Flutter application in Android Studio. These plugins provide a template to create a Flutter application, give an option to run and debug Flutter application in the Android Studio itself. 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 Dart plugin as below screen. Click yes to proceed.
Step 9.3: Restart the Android Studio.

Conclusion:

In this experiment, we successfully installed and configured the Flutter environment for developing cross-platform applications. The process involved setting up the necessary software such as Flutter SDK, Android Studio, and Visual Studio Code. After installation, the configuration was completed by setting environment variables, ensuring proper device setup, and running a basic Flutter application to verify the environment's functionality. This setup allows for efficient app development targeting multiple platforms, including Android, iOS, web, and desktop, using a single codebase. With the environment ready, developers can now proceed to build, test, and deploy Flutter applications seamlessly.

Name:Ajinkya Sunil Patil
D15B/42
Lab-mpl

Experiment 02: 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.

Factors use in my code:

Scaffold: Provides the basic structure for the UI, including the app bar, body, and navigation buttons.

AppBar: The navigation bar at the top with options like Home, All, and Settings.

TextButton.icon: Displays a button with an icon and a label in the app bar.

Container: Used for structuring and styling the display area and buttons.

GestureDetector: Wraps widgets (such as buttons) and adds touch detection, calling a function when the user interacts with it.

GridView.builder: Dynamically creates a grid layout for the calculator buttons.

FaIcon: Displays Font Awesome icons, providing better customization options for icons.

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.

Code:

```
import 'package:flutter/material.dart';
import 'package:font_awesome_flutter/font_awesome_flutter.dart';
import 'all_calculations_screen.dart'; // Import the AllCalculatorsScreen

class HomeScreen extends StatefulWidget {
  @override
  _HomeScreenState createState() => _HomeScreenState();
}
class _HomeScreenState extends State<HomeScreen> {
  bool isScientific = false; // Toggle between basic and scientific modes
  String expression = ""; // Holds the current input or result
  // Handle button presses
  void onButtonPressed(String value) {
    setState(() {
      if (value == "C") {
        expression = ""; // Clear expression
      } else if (value == "=") {
        try {
          expression = evaluateExpression(expression); // Calculate result
        } catch (e) {
          expression = "Error"; // Handle calculation errors
        }
      } else {
        expression += value; // Append button value to expression
      }
    });
  }
  // Dummy evaluate function for simplicity
  String evaluateExpression(String expr) {
    try {
      return (double.parse(expr)).toString(); // Convert expression to number
    } catch (e) {
      return "Error";
    }
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      backgroundColor: Colors.black,
```

```
    appBar: AppBar(
      backgroundColor: Colors.grey[900],
      title: Text("Smart Calculator"), // App bar title
      actions: [
        _navBarButton("Home", FontAwesomeIcons.home, () {}), // Home button
        _navBarButton("All", FontAwesomeIcons.list, () {
          Navigator.push(context, MaterialPageRoute(builder: (context) => AllCalculatorsScreen())); // Navigate to
AllCalculatorsScreen
        }),
        _navBarButton("Settings", FontAwesomeIcons.cog, () {}), // Settings button
      ],
    ),
    body: Column(
      children: [
        _buildDisplay(), // Display for showing expression or result
        _buildToggleButton(), // Button to toggle between basic and scientific modes
        _buildButtonGrid(), // Grid of calculator buttons
      ],
    ),
  );
}
// Navigation bar buttons
Widget _navBarButton(String label, IconData icon, VoidCallback onPressed) {
  return Padding(
    padding: EdgeInsets.symmetric(horizontal: 8.0),
    child: TextButton.icon(
      onPressed: onPressed,
      icon: FaIcon(icon, color: Colors.white, size: 18), // Icon from FontAwesome
      label: Text(label, style: TextStyle(color: Colors.white, fontSize: 16)), // Button label
    ),
  );
}
// Display widget for showing the calculator expression/result
Widget _buildDisplay() {
  return Container(
    width: double.infinity, // Take full width
    padding: EdgeInsets.all(16), // Padding for spacing
    alignment: Alignment.centerRight, // Align text to the right
    height: 100, // Fixed height for display area
    decoration: BoxDecoration(
      color: Colors.grey[800], // Background color for display
```

```
      borderRadius: BorderRadius.vertical(bottom: Radius.circular(10)), // Rounded bottom corners
    ),
    child: Text(
      expression, // Show the current expression/result
      style: TextStyle(fontSize: 40, fontWeight: FontWeight.bold, color: Colors.white), // Text style for display
    ),
  );
}
// Toggle button between Basic and Scientific mode
Widget _buildToggleButton() {
  return Padding(
    padding: EdgeInsets.symmetric(vertical: 8),
    child: GestureDetector(
      onTap: () {
        setState(() {
          isScientific = !isScientific; // Toggle the mode
        });
      },
      child: Container(
        padding: EdgeInsets.symmetric(vertical: 6, horizontal: 12),
        decoration: BoxDecoration(
          color: Colors.orange, // Button color
          borderRadius: BorderRadius.circular(8), // Rounded corners
        ),
        child: Text(
          isScientific ? "Basic Mode" : "Scientific Mode", // Toggle label
          style: TextStyle(color: Colors.white, fontSize: 16), // Button text style
        ),
      ),
    ),
  );
}
// Grid of calculator buttons
Widget _buildButtonGrid() {
  // Basic calculator buttons
  List<String> basicButtons = [
    "7", "8", "9", "C",
    "4", "5", "6", "/",
    "1", "2", "3", "*",
    "0", ".", "=", "+",
    "-", "(", ")", "%"
```

Name:Ajinkya Sunil Patil
D15B/42
Lab-mpl

Experiment 03: To Include Icons, Images, and Fonts in a 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.

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.

#EXECUTION

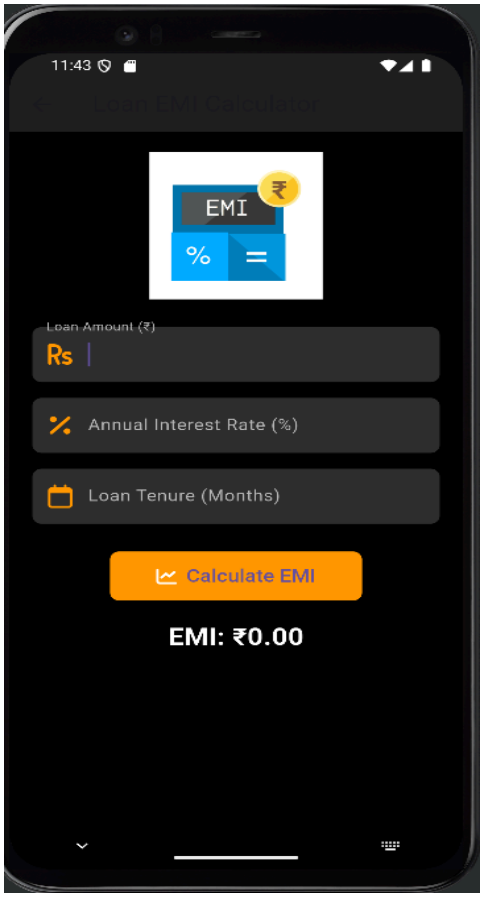
Code:-

```
import 'package:flutter/material.dart';
import 'package:font_awesome_flutter/font_awesome_flutter.dart'; // ✅ Import FontAwesome Icons
import 'dart:math'; // ✅ Importing for EMI calculations

class LoanEmiCalculator extends StatefulWidget {
  @override
  _LoanEmiCalculatorState createState() => _LoanEmiCalculatorState();
}

class _LoanEmiCalculatorState extends State<LoanEmiCalculator> {
  final TextEditingController loanAmountController = TextEditingController();
  final TextEditingController interestRateController = TextEditingController();
  final TextEditingController tenureController = TextEditingController();

  double emiResult = 0.0;

  void calculateEMI() {
    double P = double.tryParse(loanAmountController.text) ?? 0.0;
    double annualRate = double.tryParse(interestRateController.text) ?? 0.0;
    double tenureMonths = double.tryParse(tenureController.text) ?? 0.0;

    if (P == 0 || annualRate == 0 || tenureMonths == 0) {
      setState(() {
        emiResult = 0.0;
      });
      return;
    }

    double r = (annualRate / 12) / 100; // Monthly interest rate
    double n = tenureMonths; // Loan tenure in months

    double emi = (P * r * pow((1 + r), n)) / (pow((1 + r), n) - 1);

    setState(() {
```

```
    emiResult = emi;
  });
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: Text("Loan EMI Calculator"),
      backgroundColor: Colors.grey[900],
    ),
    backgroundColor: Colors.black,
    body: SingleChildScrollView(
      child: Padding(
        padding: const EdgeInsets.all(20.0),
        child: Column(
          crossAxisAlignment: CrossAxisAlignment.start,
          children: [
            // ✅ EMI Banner Image
            Center(
              child: Image.asset(
                "assets/images/EMI-Calculator.png", // Make sure this image exists
                height: 150,
                fit: BoxFit.cover,
              ),
            ),
            SizedBox(height: 20),

            _buildTextField(loanAmountController, "Loan Amount (₹)", FontAwesomeIcons.rupeeSign),
            _buildTextField(interestRateController, "Annual Interest Rate (%)", FontAwesomeIcons.percent),
            _buildTextField(tenureController, "Loan Tenure (Months)", FontAwesomeIcons.calendar),

            SizedBox(height: 20),
            Center(
              child: ElevatedButton.icon(
                onPressed: calculateEMI,
                style: ElevatedButton.styleFrom(
                  backgroundColor: Colors.orange,
                  padding: EdgeInsets.symmetric(horizontal: 40, vertical: 12),
                  shape: RoundedRectangleBorder(
                    borderRadius: BorderRadius.circular(8),
                  ),
                ),
                icon: Icon(FontAwesomeIcons.chartLine, color: Colors.white), // ✅ EMI Calculation Icon
                label: Text("Calculate EMI", style: TextStyle(fontSize: 18)),
              ),
            ),
            SizedBox(height: 20),
            Center(
              child: Text(
                "EMI: ₹${emiResult.toStringAsFixed(2)}",
                style: TextStyle(fontSize: 24, color: Colors.white, fontWeight: FontWeight.bold),
              ),
            ),
          ],
```

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

 Widget _buildTextField(TextEditingController controller, String label, IconData icon) {
  return Padding(
    padding: const EdgeInsets.symmetric(vertical: 8.0),
    child: TextField(
      controller: controller,
      keyboardType: TextInputType.number, // Ensures numeric input
      style: TextStyle(color: Colors.white), // Text color white to match dark theme
      decoration: InputDecoration(
        labelText: label,
        labelStyle: TextStyle(color: Colors.grey[400]), // Label text color
        prefixIcon: Icon(icon, color: Colors.orange), // Icon color and icon from FontAwesome
        filled: true,
        fillColor: Colors.grey[850], // Background color of the input field
        border: OutlineInputBorder(
          borderRadius: BorderRadius.circular(8),
          borderSide: BorderSide.none, // No border around the input
        ),
      ),
    ),
  );
 }
}
```

Output:



Experiment 4

Aim:To Create an Interactive Form Using Form Widgets

Interactive forms are essential in modern web applications as they allow users to input and submit data to the server for processing. These forms typically include a variety of form widgets, which enhance user engagement, making the form more dynamic and interactive. Form widgets include text boxes, buttons, checkboxes, radio buttons, dropdown lists, and more. In this experiment, the objective is to understand the importance of form widgets and how to create an effective and interactive form by using them.

What are Form Widgets?

Form widgets are HTML elements or controls used in web forms to enable users to input data in a structured way. They are the interface between the user and the backend system, helping to collect user inputs in a variety of formats (text, numbers, dates, selections, etc.). By using different form widgets, users can interact with a form in different ways, providing a wide range of inputs such as typing, selecting, or checking options.

Types of Form Widgets:

1. **Text Input Fields**: Text input fields are used for collecting short and long text information from the user. There are several variations, such as:
    - **Single-line text input** (`<input type="text">`) for basic text.
    - **Password fields** (`<input type="password">`) to hide the entered text for security.
    - **Email fields** (`<input type="email">`) to ensure proper email format validation.
    - **Textarea** (`<textarea>`) for longer multi-line text input.
2. **Radio Buttons**: Radio buttons are used when you want the user to select one option from a group of predefined choices. It ensures that only one option can be selected at a time, making it ideal for situations like selecting a gender, a payment method, or any other single-choice question.
3. **Checkboxes**: Checkboxes allow users to select one or multiple options from a list. For example, if you want the user to choose their hobbies, checkboxes enable them to select multiple hobbies from a list.
4. **Dropdown Menus (Select Boxes)**: Dropdown menus (`<select>`) allow users to choose one option from a drop-down list. They are useful when there are too many options to display on the screen at once, saving space while keeping the user experience smooth.
5. **Buttons**: Buttons in a form are used to trigger an action, typically to submit the data. There are different types of buttons:
    - **Submit button** (`<button type="submit">`) triggers form submission.
    - **Reset button** (`<button type="reset">`) clears all form fields to their initial state.
    - **Custom action buttons** can perform specific tasks, such as navigating to another page or triggering dynamic behavior through JavaScript.
6. **Date Pickers**: Date input fields allow users to pick a date from a calendar. Modern browsers provide a user-friendly date picker interface when using `<input type="date">`, ensuring consistency in date formatting.

Importance of Interactive Forms:

Forms are a primary means of communication between a user and a web application. An interactive form offers the following advantages:

- **User Engagement**: Interactive forms make it easy and enjoyable for users to enter information, reducing the chances of form abandonment.
- **Error Prevention**: Real-time form validation helps users correct their input as they type, preventing errors from being submitted and improving data accuracy.
- **Data Collection Efficiency**: Form widgets simplify complex data collection processes by using specialized inputs (such as date pickers, sliders, or dropdowns) for specific data types. This ensures that the collected data is structured, valid, and usable.

Form Validation:

Form validation ensures that the data provided by users is accurate, complete, and in the required format before submission. It can be achieved through HTML5 attributes or JavaScript. Validation checks include:

- **Required fields**: Certain fields must be filled before the form is submitted (e.g., name, email).
- **Pattern validation**: Using regular expressions, inputs can be validated for specific formats (e.g., phone numbers, email addresses).
- **Range validation**: Numeric inputs can be restricted to certain ranges (e.g., age between 18 and 60).
- **Real-time feedback**: JavaScript allows form fields to display validation messages as the user is typing, improving user experience by guiding them to correct errors immediately.

Enhancing User Experience:

User experience (UX) in forms is critical for achieving high completion rates. Several principles help enhance the UX:

- **Clear Labels and Instructions**: Labels should clearly describe what is expected in each field, while instructions guide the user through the process. For example, input placeholders or tooltips can provide hints about acceptable values.
- **Responsive Layout**: Forms should be designed to adjust to various screen sizes, making them accessible on mobile devices, tablets, and desktops. CSS media queries can help ensure proper form scaling and alignment across devices.
- **Auto-fill and Auto-complete**: Modern web browsers offer auto-fill suggestions based on users' previously entered information. This reduces the time users spend on forms, especially when entering commonly used information like name, address, or email.
- **Form Progression**: For lengthy forms, splitting them into multiple sections with a clear progression bar helps users navigate through and complete the form. This makes the form less overwhelming and allows users to track their progress.

Dynamic Forms:

Dynamic forms are forms that change their appearance or behavior based on user interactions. For example:

- Showing or hiding sections of a form based on user input (e.g., selecting "Other" from a dropdown could trigger an additional text field for more details).
- Pre-filling certain fields based on previous inputs or user data.
- Displaying success or error messages after submission to confirm whether the form was completed successfully or if corrections are needed.

Accessibility and Inclusivity:

Accessibility is critical when designing forms to ensure all users, including those with disabilities, can interact with and complete the form:

- **Labels for Inputs**: Every form input must have an associated label to make the input field identifiable for screen readers.
- **Keyboard Navigation**: Users should be able to navigate through form fields using the keyboard (e.g., using the "Tab" key to move between fields).
- **ARIA Attributes**: Adding ARIA (Accessible Rich Internet Applications) attributes improves the accessibility of dynamic content and form validation, helping users with disabilities interact with the form more effectively.

Conclusion:

Creating an interactive form using form widgets is crucial for user interaction and data collection in web applications. By utilizing various form widgets such as text fields, buttons, checkboxes, and radio buttons, we can design intuitive and user-friendly forms. Additionally, incorporating validation, responsiveness, and accessibility ensures the form provides a seamless experience for all users, resulting in accurate data collection and high completion rates. Interactive forms are an essential part of modern web design, bridging the gap between users and the application backend in an efficient and effective way.

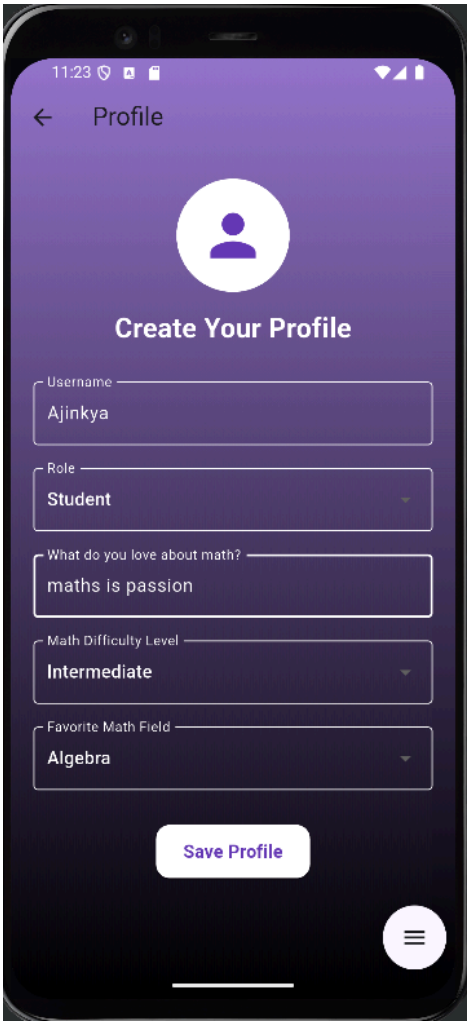
CODE:

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

class ProfileScreen extends StatefulWidget {
  @override
  _ProfileScreenState createState() => _ProfileScreenState();
}

class _ProfileScreenState extends State<ProfileScreen> {
  final _formKey = GlobalKey<FormState>();

  final TextEditingController _usernameController = TextEditingController();
  final TextEditingController _mathInterestController = TextEditingController();

  String role = "Student";
  String difficultyLevel = "Intermediate";
  String favoriteMathField = "Algebra";

  final List<String> difficultyLevels = ["Beginner", "Intermediate", "Advanced"];
  final List<String> mathFields = ["Algebra", "Geometry", "Calculus", "Trigonometry", "Statistics"];

  @override
  void dispose() {
    _usernameController.dispose();
    _mathInterestController.dispose();
    super.dispose();
  }

  void _showSuccessMessage() {
    ScaffoldMessenger.of(context).showSnackBar(
      SnackBar(
        content: Text("Profile Updated Successfully!", style: TextStyle(fontSize: 16)),
        backgroundColor: Colors.green,
        duration: Duration(seconds: 2),
      ),
    );
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      extendBodyBehindAppBar: true,
      appBar: AppBar(
        title: Text("Profile"),
        backgroundColor: Colors.transparent,
        elevation: 0,
      ),
      body: Container(
        width: double.infinity,
        decoration: BoxDecoration(
          gradient: LinearGradient(
            begin: Alignment.topCenter,
            end: Alignment.bottomCenter,
            colors: [Colors.deepPurple.shade300, Colors.black],
          ),
        ),
        child: Center(
          child: SingleChildScrollView(
            padding: EdgeInsets.all(20.0),
            child: Form(
              key: _formKey,
              child: Column(
                crossAxisAlignment: CrossAxisAlignment.center,
                children: [
                  // 🟢 User Avatar
                  CircleAvatar(
                    radius: 50,
                    backgroundColor: Colors.white,
                    child: Icon(Icons.person, size: 60, color: Colors.deepPurple),
```

```
              ),

              SizedBox(height: 15),

              // Title
              Text(
                "Create Your Profile",
                style: TextStyle(fontSize: 24, fontWeight: FontWeight.bold, color: Colors.white),
              ),

              SizedBox(height: 30),

              _buildTextField("Username", _usernameController),

              SizedBox(height: 20),

              _buildDropdown("Role", ["Student", "Teacher"], role, (value) => setState(() => role = value!)),

              SizedBox(height: 20),

              _buildTextField("What do you love about math?", _mathInterestController),

              SizedBox(height: 20),

              _buildDropdown("Math Difficulty Level", difficultyLevels, difficultyLevel, (value) => setState(() => difficultyLevel
= value!)),

              SizedBox(height: 20),

              _buildDropdown("Favorite Math Field", mathFields, favoriteMathField, (value) => setState(() => favoriteMathField =
value!)),

              SizedBox(height: 30),

              Center(
                child: ElevatedButton(
                  onPressed: () {
                    if (_formKey.currentState!.validate()) {
                      print("Username: ${_usernameController.text}, Role: $role, Interest: ${_mathInterestController.text}, Level:
$difficultyLevel, Favorite: $favoriteMathField");
                      _showSuccessMessage(); // Show success message after saving
                    }
                  },
                  style: ElevatedButton.styleFrom(
                    backgroundColor: Colors.white,
                    shape: RoundedRectangleBorder(borderRadius: BorderRadius.circular(12)),
                    padding: EdgeInsets.symmetric(horizontal: 24, vertical: 12),
                  ),
                  child: Text("Save Profile", style: TextStyle(fontSize: 16, fontWeight: FontWeight.bold, color: Colors.deepPurple)),
                ),
              ),
            ],
          ),
        ),
      ),
    ),
  );
 }

 Widget _buildTextField(String label, TextEditingController controller) {
  return TextFormField(
    controller: controller,
    style: TextStyle(color: Colors.white),
    decoration: InputDecoration(
      labelText: label,
      labelStyle: TextStyle(color: Colors.white),
      enabledBorder: OutlineInputBorder(borderSide: BorderSide(color: Colors.white)),
      focusedBorder: OutlineInputBorder(borderSide: BorderSide(color: Colors.white, width: 2)),
```

```
      ),
      validator: (value) => value!.isEmpty ? "Enter $label" : null,
    );
  }

  Widget _buildDropdown(String label, List<String> items, String selectedValue, Function(String?) onChanged) {
    return DropdownButtonFormField<String>(
      dropdownColor: Colors.grey[900],
      value: selectedValue,
      decoration: InputDecoration(
        labelText: label,
        labelStyle: TextStyle(color: Colors.white),
        enabledBorder: OutlineInputBorder(borderSide: BorderSide(color: Colors.white)),
        focusedBorder: OutlineInputBorder(borderSide: BorderSide(color: Colors.white, width: 2)),
      ),
      items: items.map((item) => DropdownMenuItem(value: item, child: Text(item, style: TextStyle(color:
Colors.white)))).toList(),
      onChanged: onChanged,
    );
  }
}
```

OUTPUT:



Name:Ajinkya Sunil Patil
D15B/42
Lab-mpl

Experiment 5

Aim:Applying Navigation, Routing, and Gestures in a Flutter App

Introduction:
Flutter is a powerful framework for building cross-platform mobile applications that can run on Android, iOS, and the web. One of the key features that contribute to Flutter's ease of use is its built-in support for navigation, routing, and gestures. These three components are essential for creating user-friendly and interactive apps, providing a seamless experience for users as they move between screens and interact with various elements.

In this writeup, we will explore the theoretical aspects of navigation, routing, and gestures in Flutter, covering their concepts, importance, and how they are typically implemented in Flutter apps.

Theory:

1. Navigation in Flutter

Navigation refers to the process of transitioning between different screens or views (called widgets in Flutter) within an app. Most mobile apps consist of multiple screens such as a home screen, settings, profile, and so on. Users move between these screens by navigating from one to another.

In Flutter, navigation is typically managed using the **Navigator** class. The Navigator is a widget that manages a stack of pages, or routes, and provides methods for navigating between them. When a user moves from one page to another, the current page is pushed onto the stack, and when the user goes back, the page is popped from the stack.

Flutter uses a stack-based navigation system, where the stack follows the last-in, first-out (LIFO) principle. The stack's structure allows for easy transitions between screens and keeps track of the user's navigation history. Common methods used in navigation are:

- **push**: Adds a new route to the stack.
- **pop**: Removes the current route from the stack and returns to the previous one.
- **pushReplacement**: Replaces the current route with a new one, removing the existing one.

Importance of Navigation

Navigation is essential in any app with multiple screens, as it allows the user to move seamlessly between them. Proper navigation ensures:

- A smooth user experience.
- Consistency in app flow and screen transitions.
- Efficient resource management, as only necessary screens are loaded into memory.

Effective navigation in an app also contributes to user retention and satisfaction, as it avoids confusion and makes the app easier to use.

---

2. Routing in Flutter

Routing is the process of defining and controlling how the app responds to different navigation requests. A route represents a screen or a page in Flutter, and routing determines which route to load when navigating through the app. Flutter's routing system is flexible and can be implemented in various ways, depending on the app's complexity.

Flutter supports two primary types of routing:

- **Imperative Routing**: This is the default approach where routes are defined dynamically as needed. Developers call Navigator methods to push and pop routes, explicitly controlling the navigation.
- **Declarative Routing**: This approach involves defining routes and navigation paths at the start. It's useful in more complex apps where there is a need for predefined and predictable routing.

Types of Routing

- **Basic Routing**: In this simple form of routing, the developer manually defines routes using the Navigator's push and pop methods. Basic routing is ideal for small apps with fewer screens, as it is relatively straightforward to implement.

- **Named Routing**: In more extensive apps, it is often beneficial to use named routes, which allow developers to define routes as named identifiers. Named routing helps improve code organization and makes navigation between screens more readable and easier to maintain.

- **On-Generate Route**: Flutter provides a callback called onGenerateRoute, which is used to handle unknown routes or to perform dynamic route generation based on user actions. This approach is often used when there are complex navigation patterns or when routes need to be constructed dynamically.

- **Nested Navigation**: Flutter supports nested navigation, which allows developers to create independent navigators within different sections of an app. This is helpful in scenarios where different parts of the app require independent navigation flows (e.g., navigating within tabs).

Importance of Routing

Routing is crucial because it determines how users navigate within an app. Proper routing provides:

- Clear paths for users to move through the app's content.
- Improved code organization, especially in larger apps with complex navigation flows.
- Flexibility to handle various navigation scenarios, including deep linking and nested routes.

---

3. Gestures in Flutter

Gestures are actions that users perform on a mobile device's touchscreen, such as tapping, swiping, or dragging. Flutter provides a robust gesture detection system that allows developers to capture user interactions and respond to them accordingly.

Flutter's gesture system is based on widgets called **GestureDetectors**, which capture various gestures and provide callback functions that trigger when a gesture is detected. Some of the common gestures include:

- **Tap**: A quick touch on the screen.
- **Double-tap**: Two rapid touches on the screen.
- **Long-press**: A prolonged touch on the screen.
- **Swipe**: A quick drag across the screen, typically in a specific direction (left, right, up, or down).
- **Drag**: Moving the finger across the screen without lifting it.

GestureDetector Widget

The GestureDetector widget is the primary tool for capturing gestures in Flutter. It allows developers to define callback functions for specific gestures. For example, a button tap can be captured with the onTap callback, while a swipe action can be detected with onHorizontalDrag.

Importance of Gestures

Gestures play a vital role in modern mobile apps, as they:

- Provide an intuitive and natural way for users to interact with the app.
- Enhance the user experience by making the app feel more responsive and fluid.
- Allow developers to create dynamic interfaces with interactions such as scrolling, zooming, and dragging.

Gestures also form the backbone of many mobile interfaces, such as swipe-to-delete, pull-to-refresh, and pinch-to-zoom. Implementing gestures correctly ensures that users can navigate the app efficiently and enjoy a smooth experience.

---

4. Combining Navigation, Routing, and Gestures

In a typical Flutter app, navigation, routing, and gestures work together to provide a cohesive and intuitive user experience. For instance:

- Navigation allows users to move between different screens.
- Routing defines how the app responds to navigation requests.
- Gestures enable users to interact with the app's elements, such as tapping buttons to navigate or swiping to trigger actions.

By combining these three elements, developers can create sophisticated and user-friendly apps that offer smooth transitions, clear navigation paths, and interactive gestures. For example, users can swipe through different pages in a gallery app, tap a button to navigate to the details of a photo, and use named routing to easily return to the gallery from any screen.

Conclusion

Navigation, routing, and gestures are fundamental concepts in Flutter app development. Together, they create a seamless, intuitive, and interactive experience for users. Proper implementation of these components allows developers to build apps that are not only functional but also engaging and easy to use.

In Flutter, the navigation system, including the Navigator and routes, manages transitions between different screens. Routing ensures that users are guided through the app in a logical and efficient way, while gestures enhance interactivity by allowing users to perform actions through touch-based inputs. When these three elements are integrated into an app, they significantly contribute to its overall usability and design.

Code:

```
import 'package:flutter/material.dart';
import 'package:font_awesome_flutter/font_awesome_flutter.dart';
import 'loan_emi_calculator.dart';
import 'time_unit_calculator.dart';
import 'currency_converter.dart';
import 'unit_converter.dart'; // ✅ Import Unit Converter Screen

class AllCalculatorsScreen extends StatelessWidget {
 const AllCalculatorsScreen({super.key});

 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(
    title: Text("All Calculators"),
    backgroundColor: Colors.grey[900],
   ),
   backgroundColor: Colors.black,
   body: GridView.count(
    padding: EdgeInsets.all(12),
    crossAxisCount: 2,
    crossAxisSpacing: 12,
    mainAxisSpacing: 12,
    children: [
     _buildCalculatorButton(context, "Calculator",
       FontAwesomeIcons.calculator, Colors.orange),
     _buildCalculatorButton(context, "Time Converter & Calculations",
       FontAwesomeIcons.clock, Colors.green),
     _buildCalculatorButton(context, "Currency Converter",
       FontAwesomeIcons.moneyBill, Colors.blue),
     _buildCalculatorButton(context, "Unit Converter",
       FontAwesomeIcons.ruler, Colors.purple), // ✅ FIXED
     _buildCalculatorButton(context, "Loan EMI Calculator",
       FontAwesomeIcons.creditCard, Colors.red),
     _buildCalculatorButton(context, "Date Calculations",
       FontAwesomeIcons.calendar, Colors.teal),
    ],
   ),
  );
 }

 Widget _buildCalculatorButton(
   BuildContext context, String label, IconData icon, Color color) {
  return GestureDetector(
   // 🔹 Gesture detection for button tap
   onTap: () {
    print("$label button tapped!"); // 🔍 Debugging Print

    // 🔹 Navigation logic for each calculator
    if (label == "Unit Converter") {
     Navigator.push(context,
       MaterialPageRoute(builder: (context) => UnitConverterScreen())); // ✅ Navigate to Unit Converter Screen
    } else if (label == "Loan EMI Calculator") {
     Navigator.push(context,
       MaterialPageRoute(builder: (context) => LoanEmiCalculator())); // ✅ Navigate to Loan EMI Calculator
    } else if (label == "Time Converter & Calculations") {
     Navigator.push(context,
       MaterialPageRoute(builder: (context) => TimeUnitCalculator())); // ✅ Navigate to Time Unit Calculator
    } else if (label == "Currency Converter") {
     Navigator.push(
       context,
       MaterialPageRoute(
         builder: (context) => CurrencyConverterScreen())); // ✅ Navigate to Currency Converter
    } else {
     ScaffoldMessenger.of(context).showSnackBar(SnackBar(
```

```
          content: Text("$label is in working state"),
          duration: Duration(seconds: 2),
        ));
      }
    },
    child: Container(
      decoration: BoxDecoration(
        color: color,
        borderRadius: BorderRadius.circular(12),
      ),
      padding: EdgeInsets.all(16),
      child: Column(
        mainAxisAlignment: MainAxisAlignment.center,
        children: [
          FaIcon(icon, size: 40, color: Colors.white),
          SizedBox(height: 10),
          Text(label,
              textAlign: TextAlign.center,
              style: TextStyle(color: Colors.white, fontSize: 16)),
        ],
      ),
    ),
  );
 }
}
```

Highlighted Sections:

- **Gesture Detection**: The `GestureDetector` is used to capture taps on the buttons.
- **Navigation**: The app navigates to the respective screen (Unit Converter, Loan EMI Calculator, etc.) based on the button tapped.

OUTPUT:

Practical-06

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:

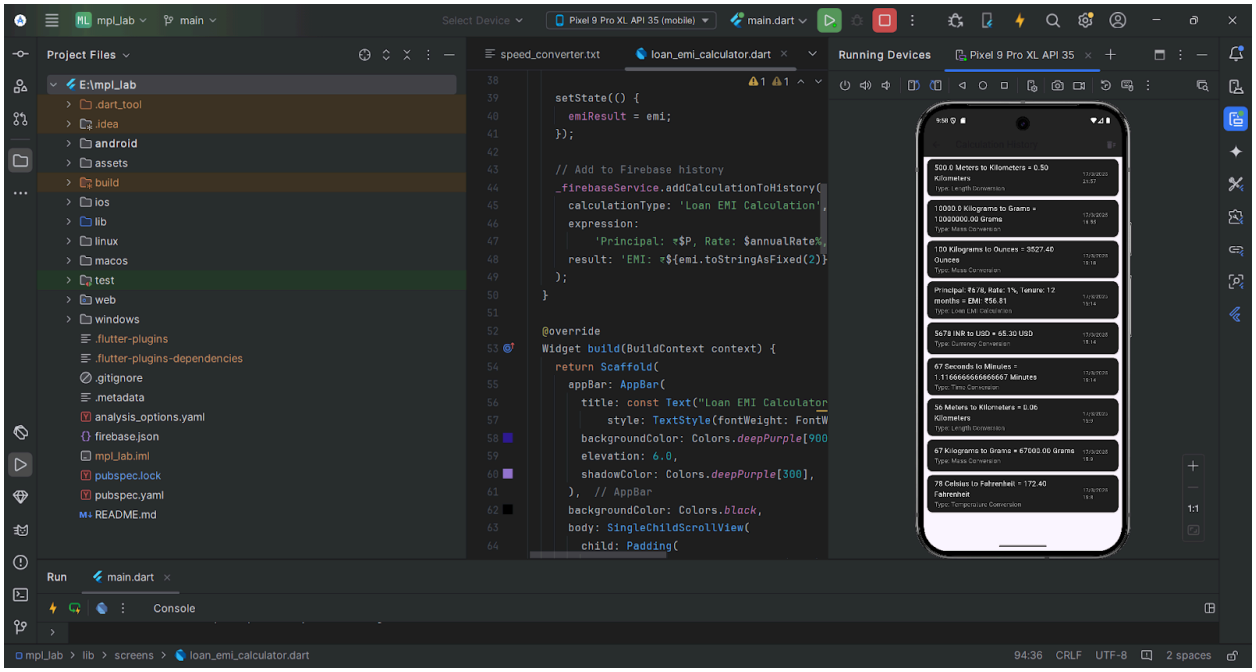
1. **Firebase Setup and Configuration:**
    - Create a Firebase project and add the Flutter app to the project.
    - Download the google-services.json (for Android) and GoogleService-Info.plist (for iOS) and add them to the respective directories in the Flutter project.
    - Add Firebase dependencies 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.

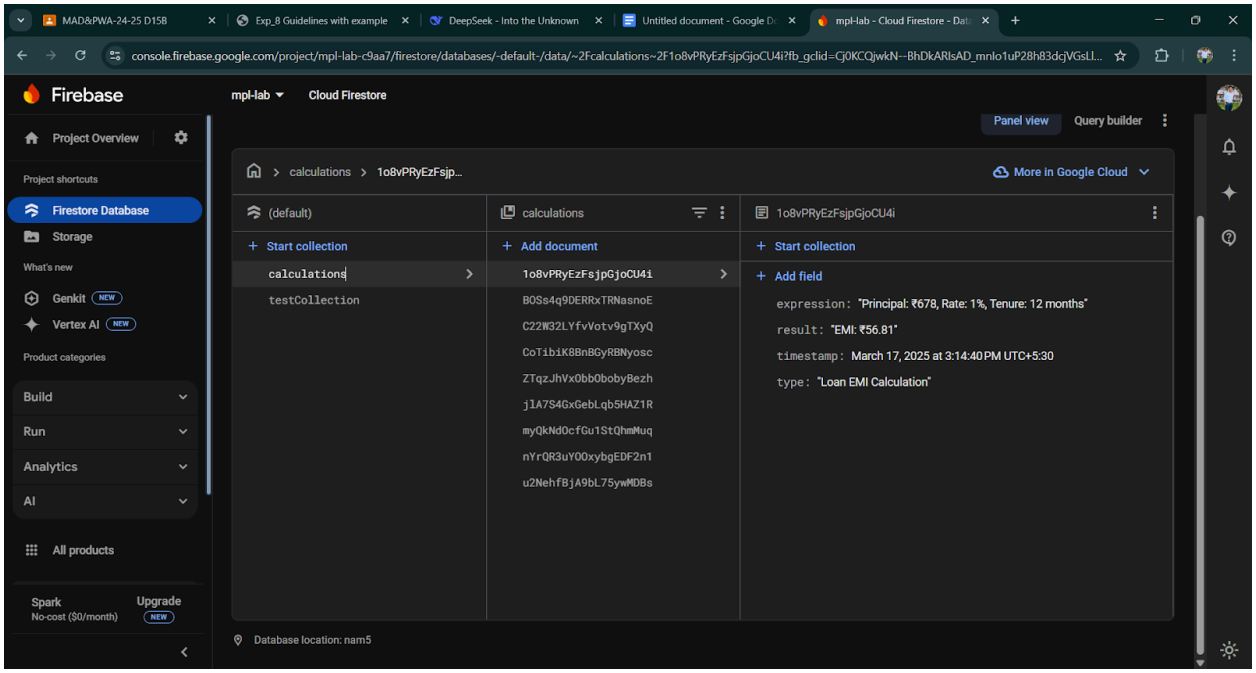
    - Update: Use update() to modify specific fields without overwriting other data.
    - Delete: Remove documents or nodes using the delete() method.
4. **Implementing Authentication:**
    - Integrate Firebase Authentication for user sign-in and registration.
    - Implement email/password authentication 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.

Name:Ajinkya Sunil Patil D15B/42





EXPERIMENT No. 7

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.

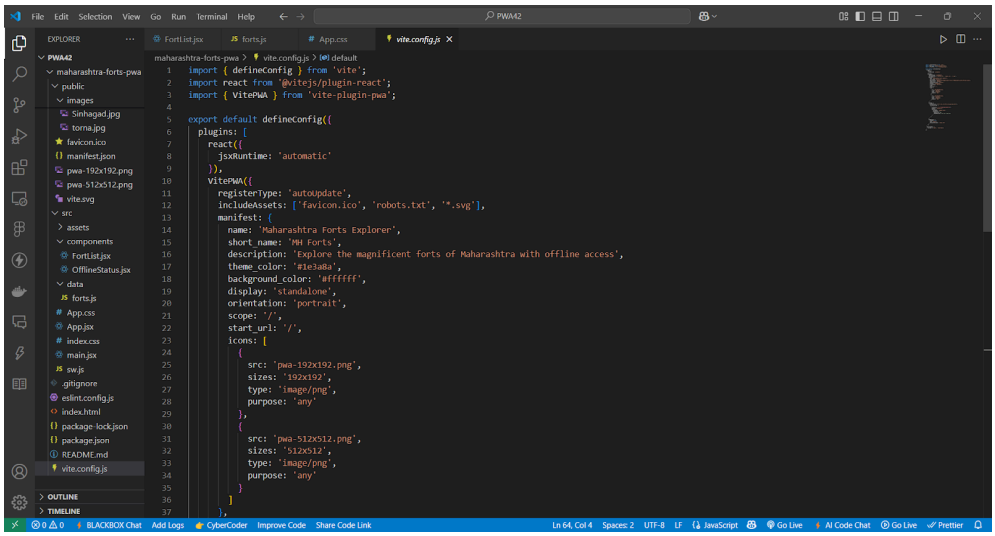
2 Configure Vite for PWA

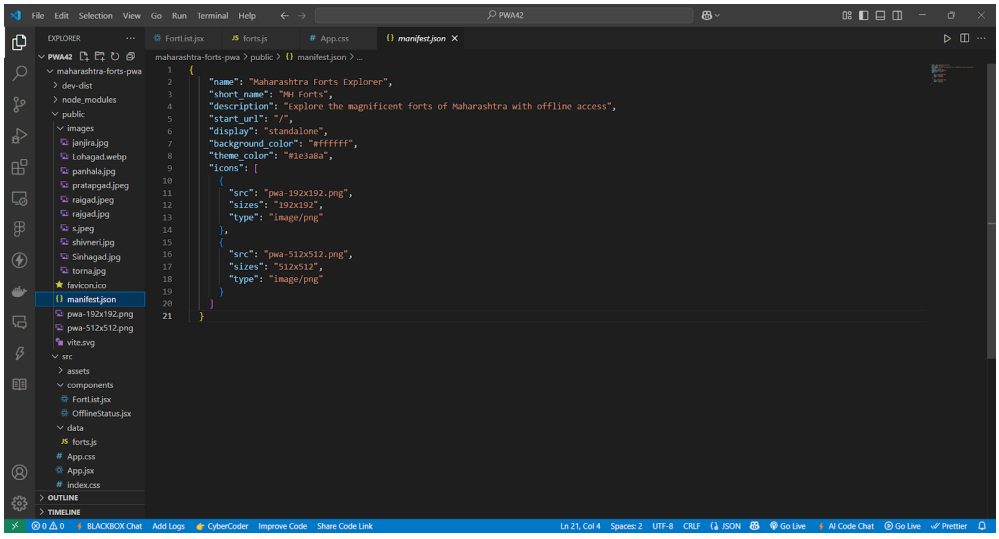
1. Open the `vite.config.js` file in your project.
2. Import and add the PWA plugin inside the `plugins` section.
3. Set up options like auto-updates, caching, and offline support.
4. Save the file and restart the Vite server.

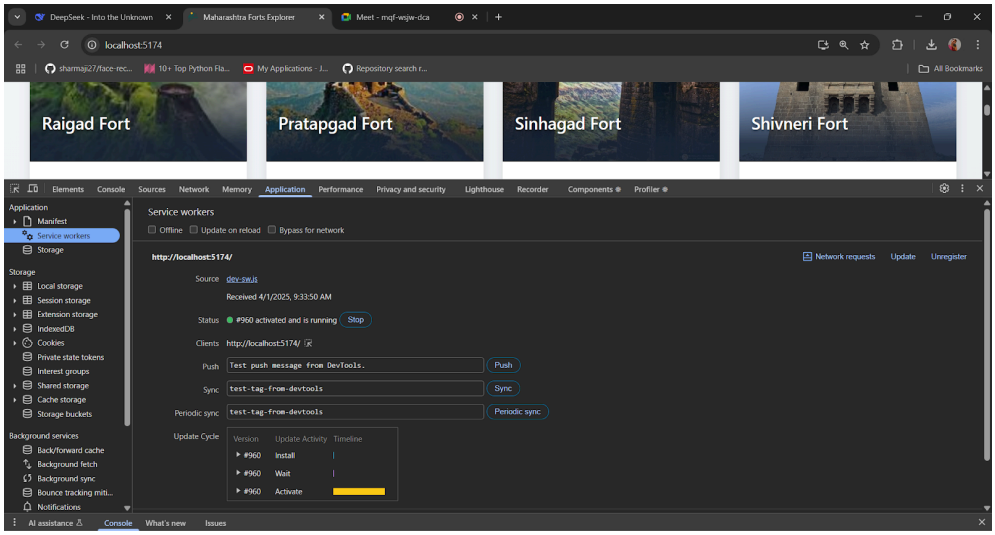
3 Add `manifest.json` File

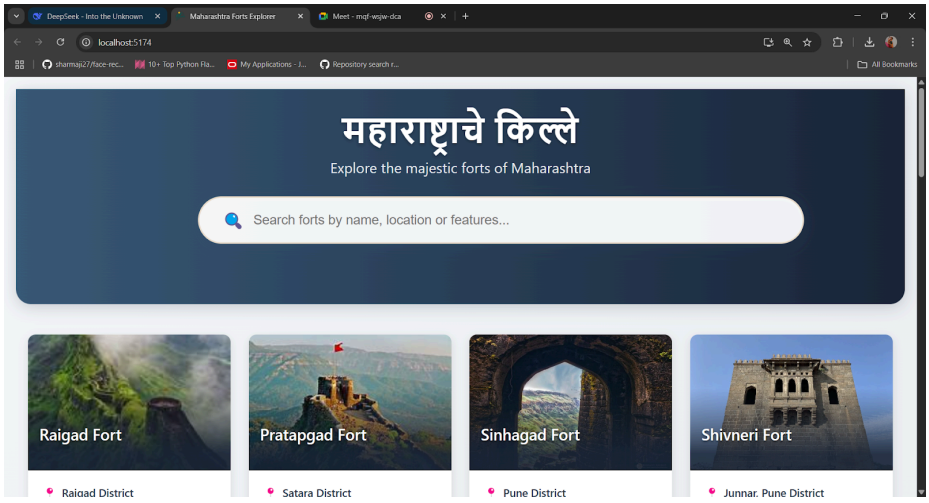
1. Go to the `public` folder in your project.
2. Create a new file named `manifest.json`.
3. Add details like the app name, icons, theme color, and display mode.
4. This file helps browsers recognize your app as installable**.**

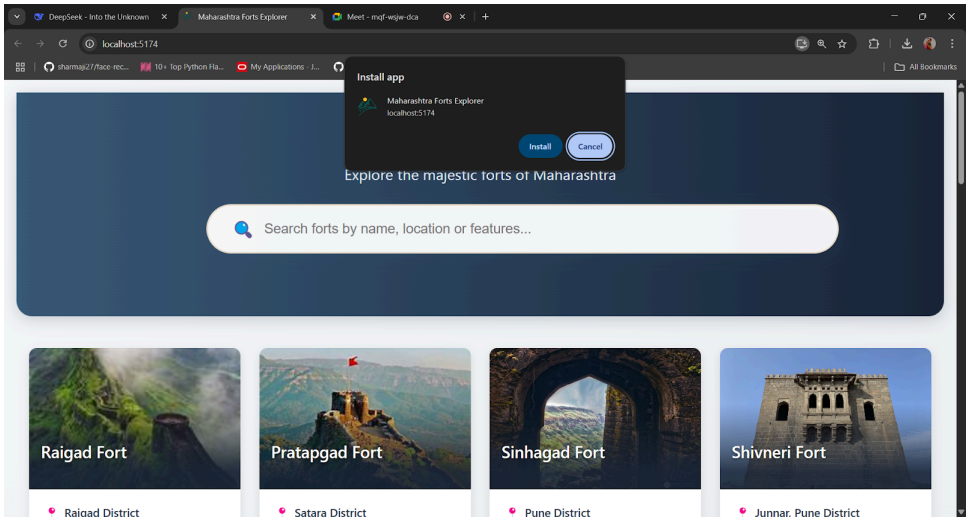
OUTPUT :




manifest.json
{
"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"
}
]
}


Maharashtra Forts Explorer
localhost:5174
Raigad Fort
Pratapgad Fort
Sinhagad Fort
Shivneri Fort
Application
Service workers
Storage
http://localhost:5174/
Push
Sync
Periodic sync


Maharashtra Forts Explorer
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

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.

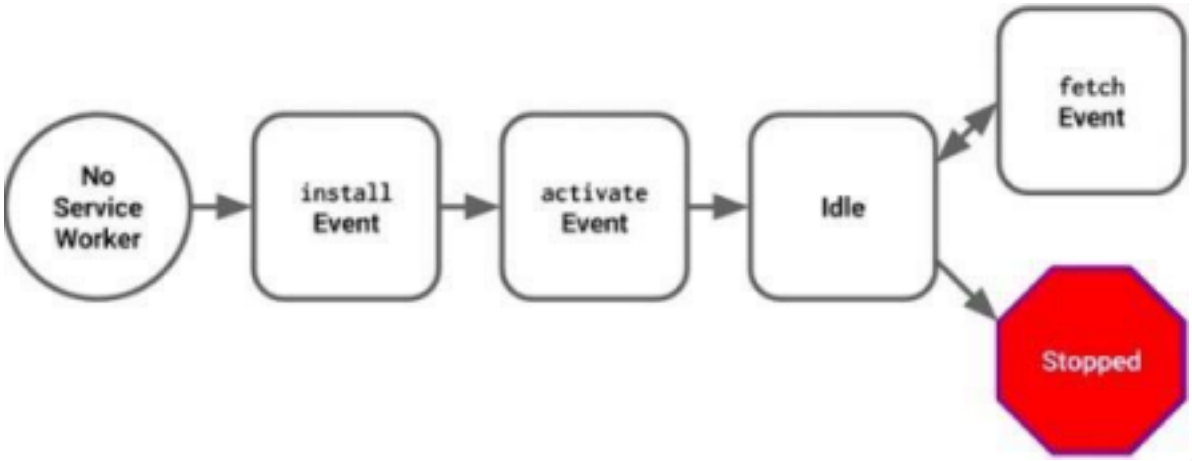
Theory:

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 :

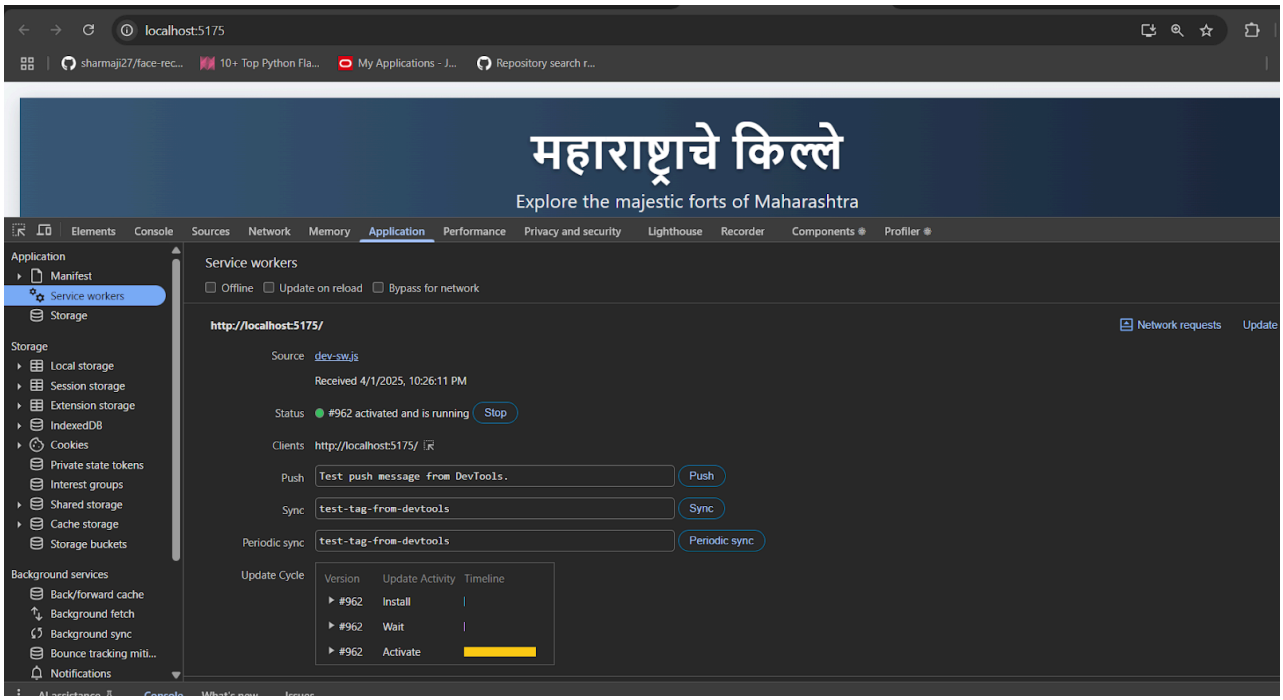
sw.js

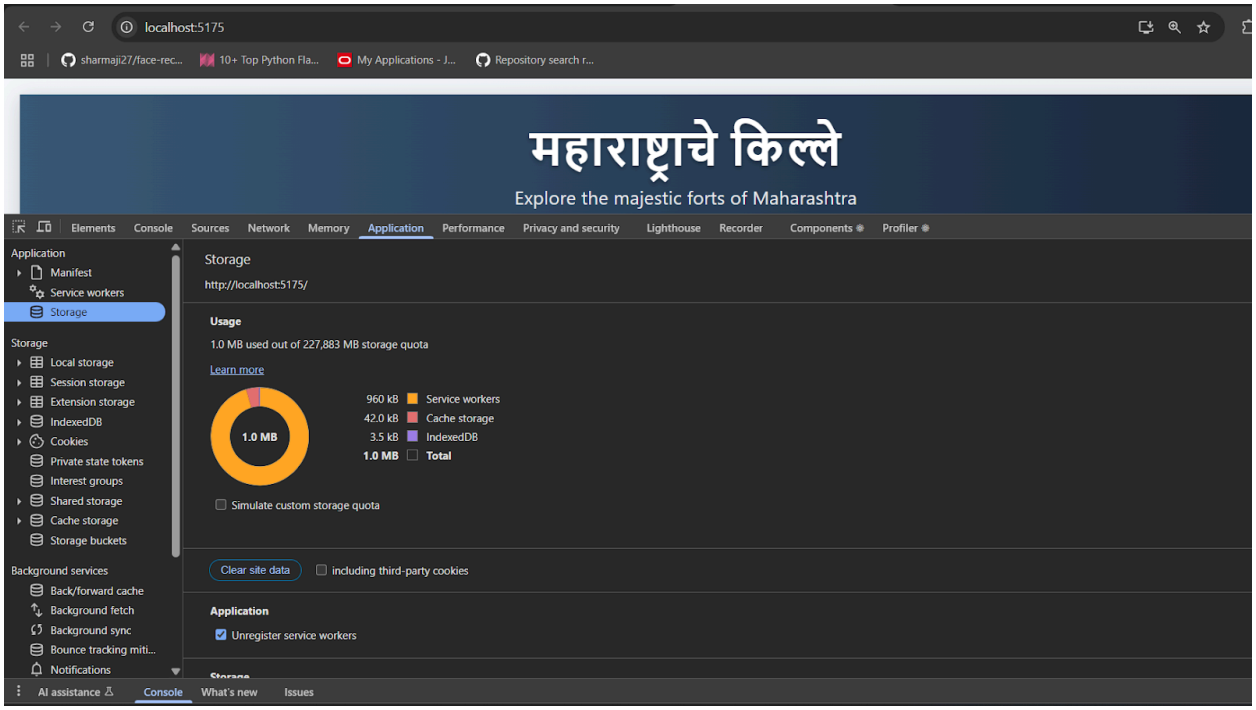
```
/**
 * Copyright 2018 Google Inc. All Rights Reserved.
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *     http://www.apache.org/licenses/LICENSE-2.0
 * Unless required 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();
          }
        })
```

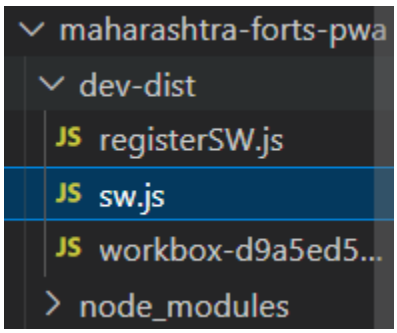
```
      .then(() => {
        let promise = registry[uri];
        if (!promise) {
          throw new Error(`Module ${uri} didn't register its module`);
        }
        return promise;
      })
    );
  };

  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",
```

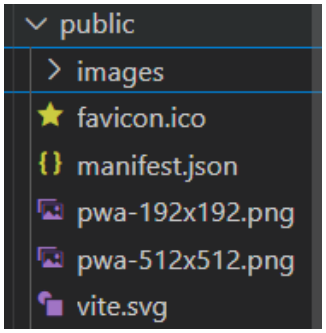
```
    "revision": "3ca0b8505b4bec776b69afdba2768812"
  }, {
    "url": "index.html",
    "revision": "0.7e7jl0981k"
  }], {});
  workbox.cleanupOutdatedCaches();
  workbox.registerRoute(new
workbox.NavigationRoute(workbox.createHandlerBoundToURL("index.html"), {
    allowlist: [/^\/$/]
  }));
  workbox.registerRoute(/\.(?:png|jpg|jpeg|svg)$/,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

}));
```

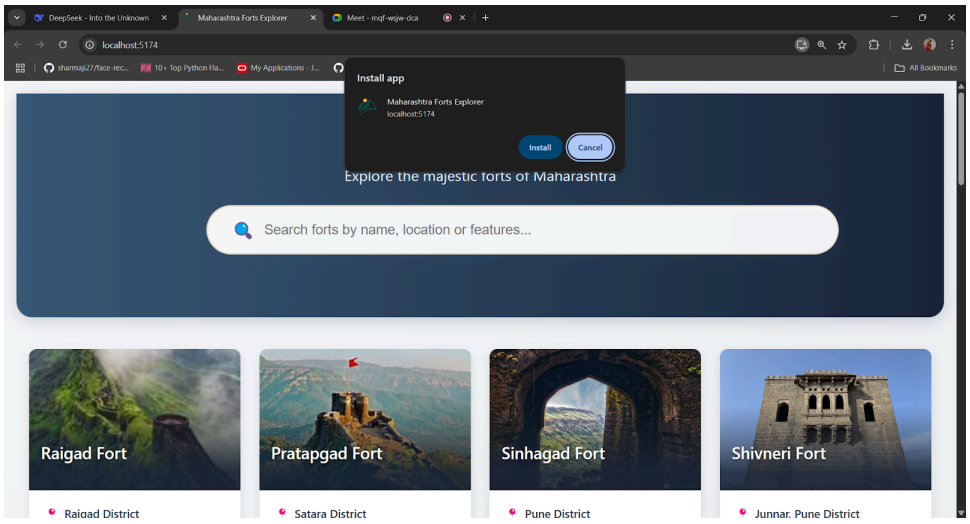
OUTPUT :

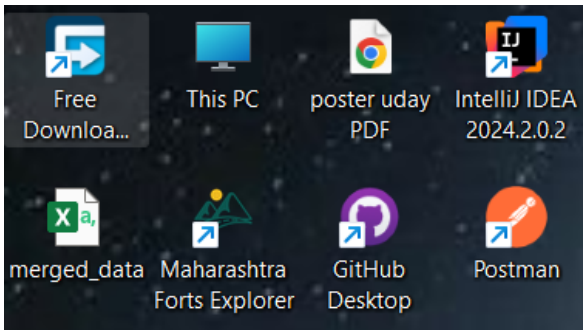



localhost:5175
महाराष्ट्राचे किल्ले
Explore the majestic forts of Maharashtra
Elements
Console
Sources
Network
Memory
Application
Performance
Privacy and security
Lighthouse
Recorder
Components
Profiler
Application
Manifest
Service workers
Storage
Storage
Local storage
Session storage
Extension storage
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

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”.

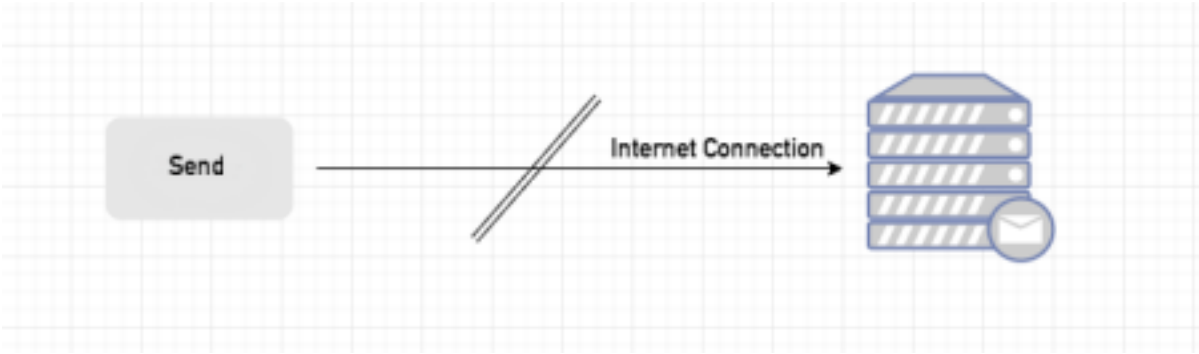
- **CacheFirst** - In this function, if the received request has cached before, the cached response is returned to the page. But if not, a new response requested from the network.
- **NetworkFirst** - In this function, firstly we can try getting an updated response from the network, if this process completed 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

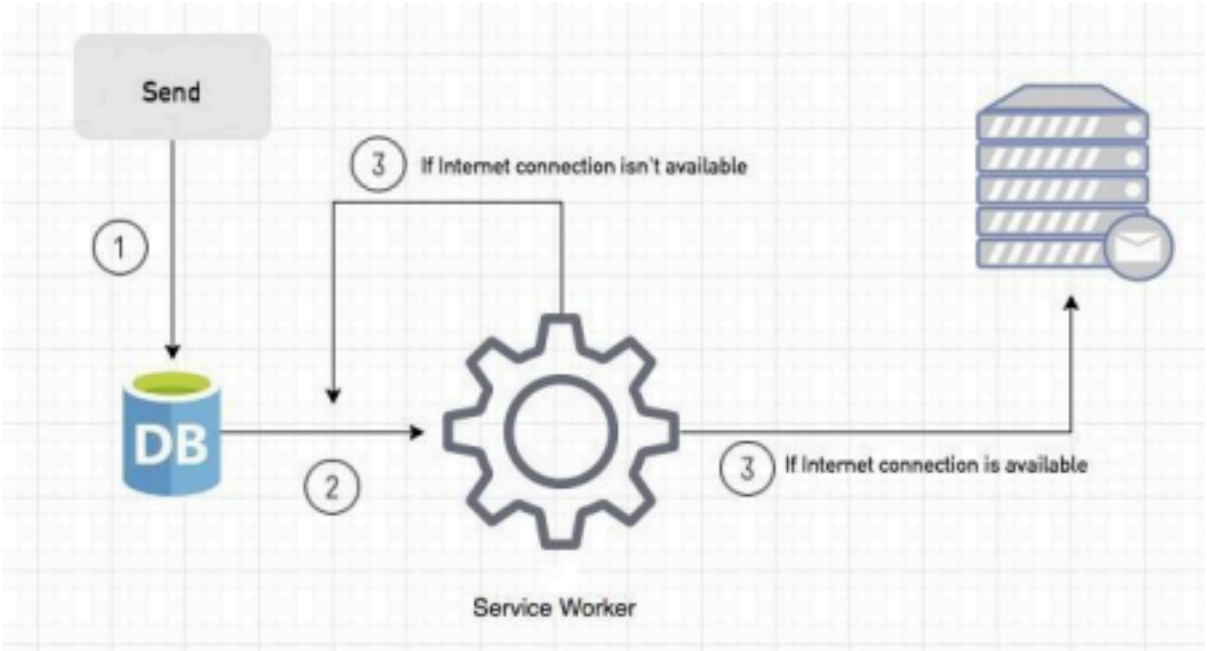
Background Sync is a Web API that is used to delay a process until the Internet connection is stable. We can adapt this definition to the real world; there is an e-mail client application that works on the browser and we want to send an email

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

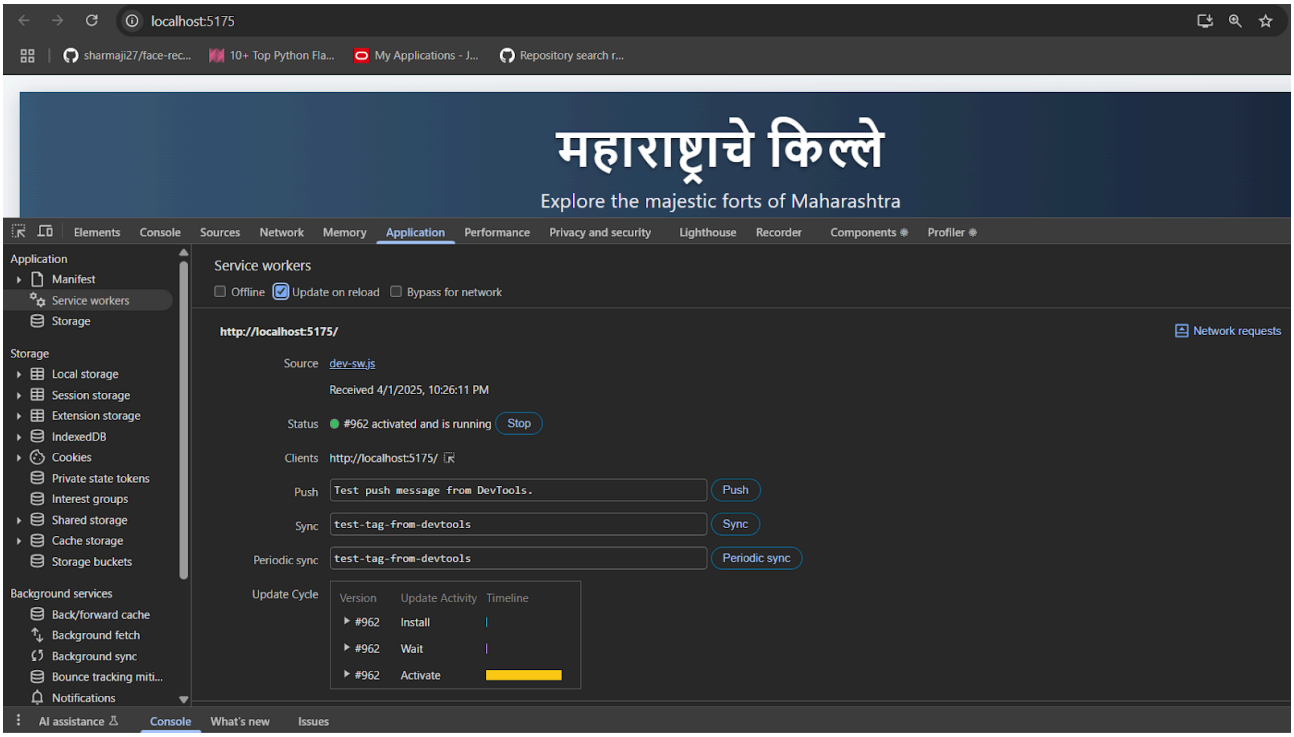
This is the event that handles push notifications that are received from the server. You can apply any method with received data.

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

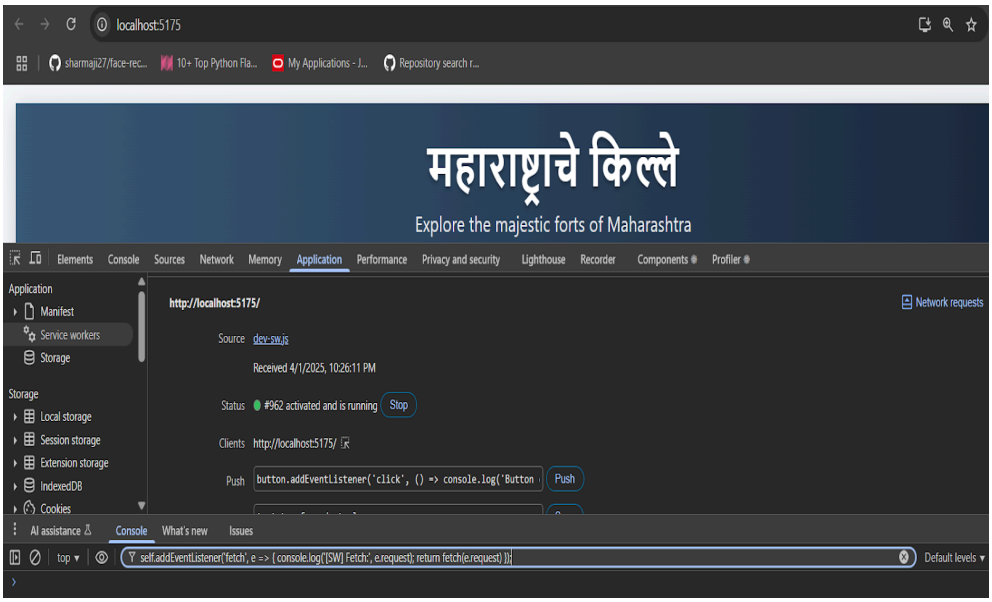
```
 * Copyright 2018 Google Inc. All Rights Reserved.
 * Licensed under the Apache License, Version 2.0 (the "License");
 * you may not use this file except in compliance with the License.
 * You may obtain a copy of the License at
 *     http://www.apache.org/licenses/LICENSE-2.0
 * Unless required 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",
```

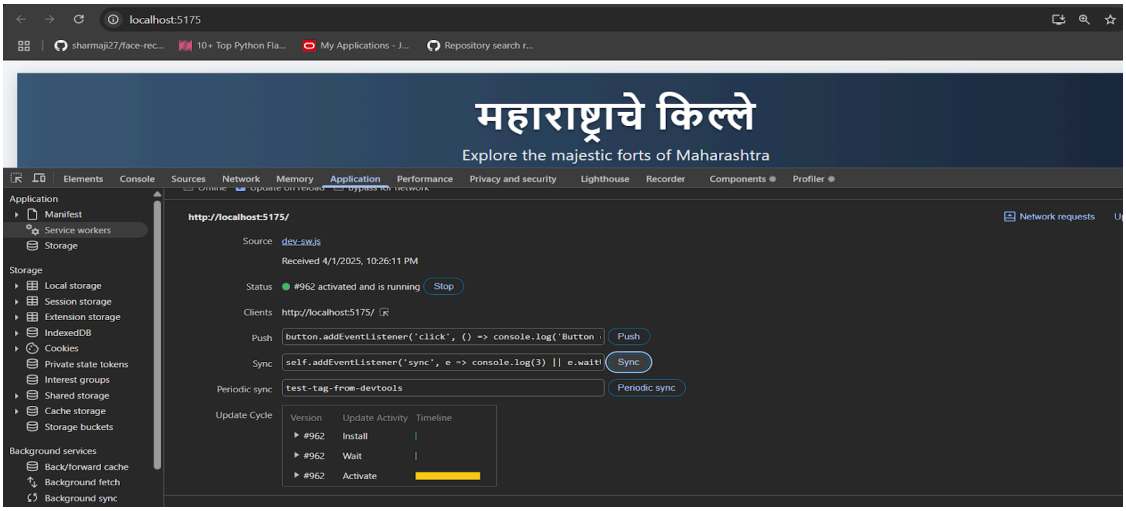
```
  "revision": "0.7e7jl0981k"
 }], {});
 workbox.cleanupOutdatedCaches();
 workbox.registerRoute(new workbox.NavigationRoute(workbox.createHandlerBoundToURL("index.html"),
{
  allowlist: [/^\/$/]
 }));
 workbox.registerRoute(/\.(?:png|jpg|jpeg|svg)$/, new workbox.CacheFirst({
  "cacheName": "images-cache",
  plugins: [new workbox.ExpirationPlugin({
   maxEntries: 50,
```

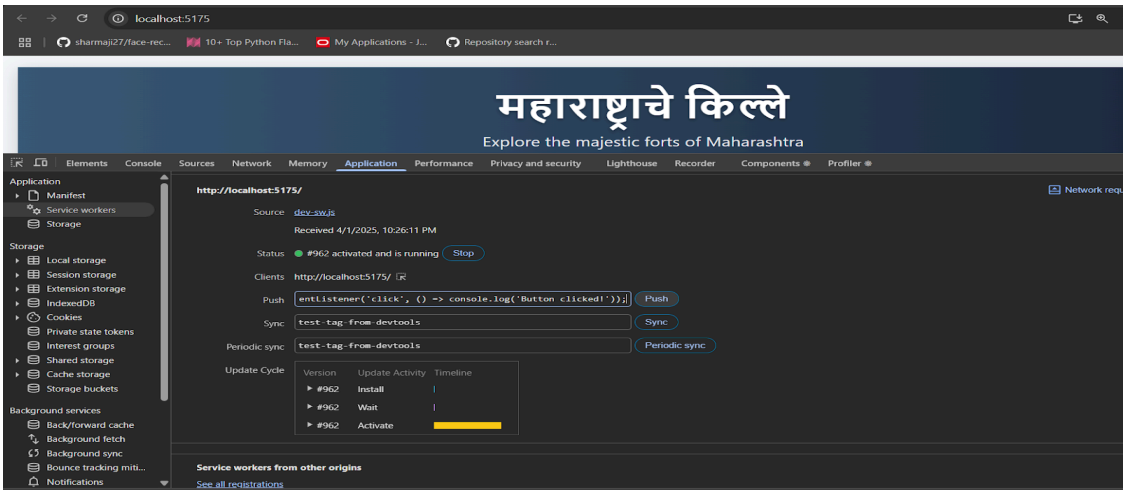
Output:

Fetch event

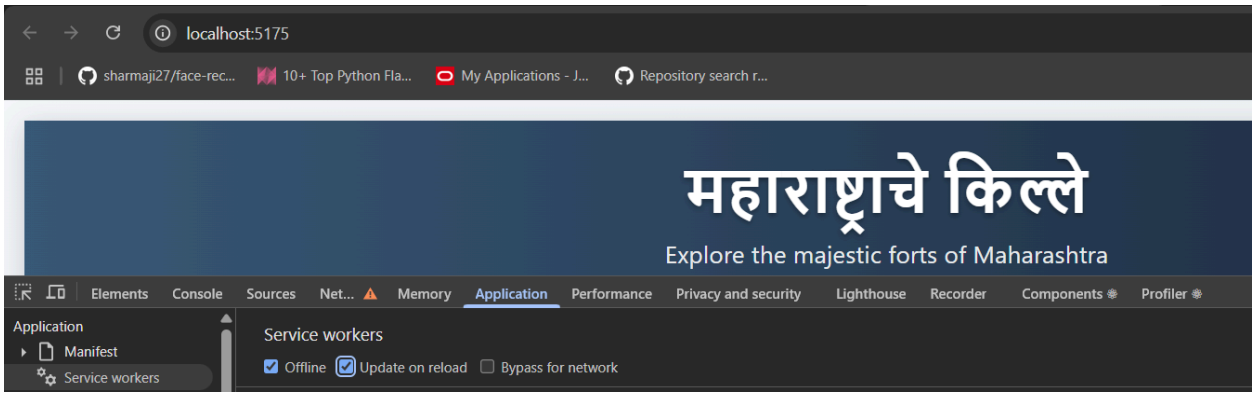


Sync event and Push event



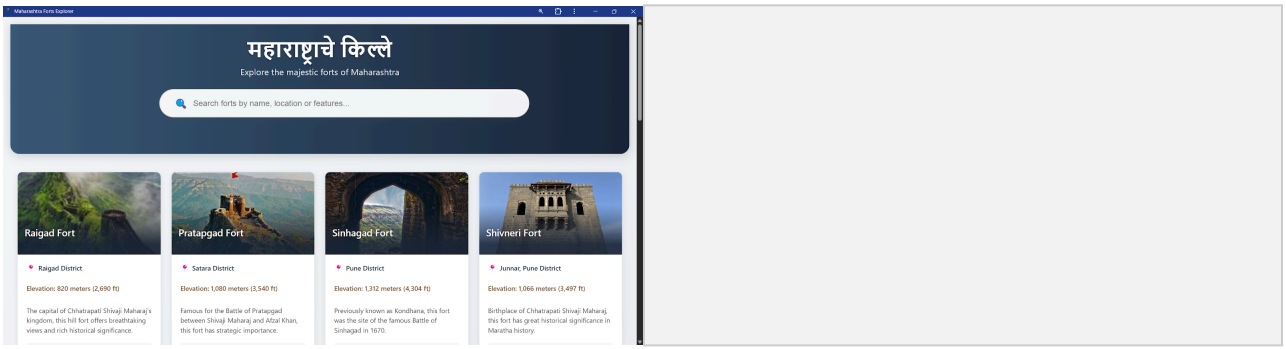


OFFLINE ACCESS



Without Network

BY Download in System



Experiment No. 10

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/ajinkyaspatil20/github_pages

Github Screenshot:
https://ajinkyaspatil20.github.io/githubs_pages/

Experiment No: 11

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.

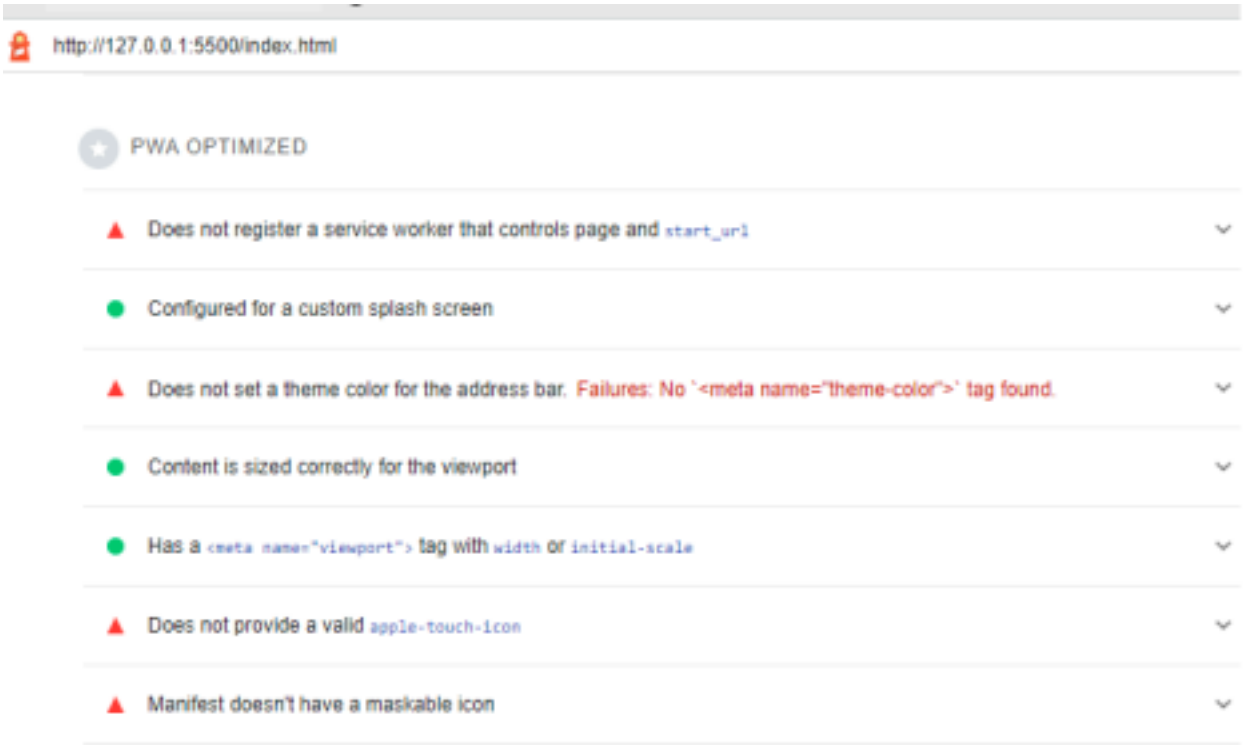
4. **Best Practices:** As any developer would know, there are a number of practices that have been deemed 'best' based on empirical data. This metric is an aggregation of many such points, including but not limited to:Use of HTTPS

   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"

      }
    ]
 }
```

Name:Ajinkya Sunil Patil **Exp_No.:11** **D15B-42**

Conclusion: Thus we successfully used google Lighthouse PWA Analysis Tool for testing the PWA functioning.

Name :- Ajinkya Sunil Patil D15B-42

AH OS

MPL Assignment 01

Q1.

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

Soln :-

Key features of flutter :-

1) Single codebase - Write one code for both Android and ios.

2) Fast performance - Uses Dart language and a 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.

Advantages of using flutter :-

1) Save Time & Effort - Single codebase for multiple platforms.

2) High Speed Development - Hot Reload features speeds up codings.

3) Cost-Effective - Reduces Development cost & time.

4) Attractive UI - provides beautiful and customizable widgets.

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

Soln:-

How Flutter Differs from traditional Approaches:

1) Single codebase - Traditional methods needs separate code for Android and ios.
2) Hot Reload - Traditional apps require full restart after changes, but flutter update instan
3) UI Rendering - Traditional apps use native components, but flutter update and rendering.
4) Performance - Flutter compiles directly to native machine code, making it faster than framework.

Why flutter is popular Among Developers:-

1) fast Development - Hot Reload and single codebase saves time.
2) Cross-platform - work on mobile, web and desktop
3) Beautiful UI - Rich, customizable widgets than layer like React Native.
4) High performance - Runs smoothly without a bridge like React Native.
5) Active community - Regular update and strong community help developers.

Q2]

a] Describe the concept of the widgets tree in flutter. Explain how widget compositions is used to build complex user interfaces.

Soln :- 1) Concept of Widget Tree in Flutter :

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

2) For Example, a Scaffold widgets can have a Column widgets, which contain Text and button widgets. Changes in widgets updates the tree dynamically.

3) Widgets compositions for complex UI :-

Flutter uses small, reusable widgets to build complex UI. Instead of creating a single large UI blocks, developers combine multiple small widgets likes Rows, Columns, Containers and buttons.

e.g :-

i) A ListView can contain multiple Card widgets.

ii) A Column can hold Text, Images and Buttons.

This modular approach make the UI flexible, reusable, and easy to manage.

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

Soln :-

Commonly Used Widgets and Their Roles in a widget Tree:

i) Scaffold - basic layout structure.

ii) Appbar - Top navigation bar with title.

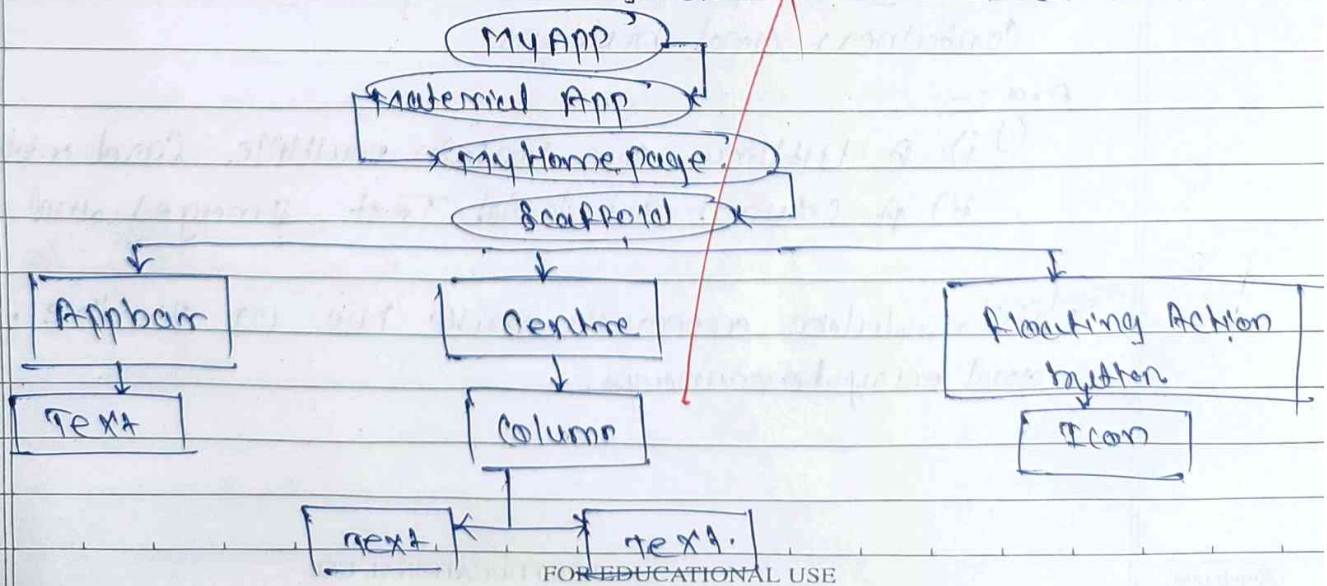
iii) Text - Simple Text on a screen

iv) Image - Show image from assets url's

v) Container - Used for styling

vi) Row - arrangement in horizontal structure

vii) Column - arrangement in vertical structure.



Q3]

a] Discuss the importance of State management in Flutter applications.

Soln :-

Importance of State Management in Flutter Application :-
State management is important because it controls how the app stores, updates and displays data when the users interfaces with it.

Why State Management is Needed?
1) Keeps UI updated - Ensure that the app reflects changes.
2) Improves performance - updates only necessary part of the UI instead of reloading
3) Manage complex data.
4) Ensure smooth user Experience.

Types of State in Flutter -

1) Local State.
2) Global State.

b] Compare and contrast the different State management approaches available in flutter such as SetState, Provider, and Riverpod, provides scenario where each approach is suitable.

Approach	How it works	When to Use.
SetState.	update UI by calling setState()	Best for small apps or single widgets.
Provider	InheritedWidget to share state across widgets	Suitable for medium-sized apps
Riverpod	Improved version of provider with better performance.	Best for large apps

Choosing the Right Approach :

→ Use setState for simple UI update
→ Use provider for moderate state
→ Use Riverpod for well-structure applications.

Q4]

a] Explain the process of integrating firebase with a flutter application. Discuss the benefits of using firebase as a backend solution.

Soln :-

Process of integrating Firebase with a flutter application.

1. Create a Firebase project.
2. Add Firebase to flutter app
3. Intall firebase packages.
4. Initialize Firebase.
5. Use firebase services.

Benefits of Using Firebase as a backend soln:-

1. Realtime Database.
2. Authentication
3. Cloud Firestore
4. Hosting & Storage
5. push Notification.

b] Highlight the Firebase services commonly used in flutter development and provide a brief overview of how data synchronization is achieved.

soln:-

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

6. Firebase Hosting
7. Firebase Analytics

How Data Synchronization is Achieved:-

1. Realtime Updates
2. Listeners & Streams
3. Offline Support.

Name:- Ajinkya Sunil Patil | D15B-42

05/05

MPL Assignment 02

1] Define progressive webapp (PWA) and explain its significance in modern web development. Discuss the key characteristics PWAs from traditional mobile apps.

Soln:-

A progressive webapp (PWA) is a type of web application that works like a mobile capp but runs in a browser

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

Difference in PWA & Traditional Apps:-

Features	PWA	Traditional App
Installation	Direct from Browser	Download from App Store.
Internet Required	Work offline with caching	Usually requires internet.
Performance	Fast with service workers	Fast but need installation.
updates	Automatic	Manual.
Development Cost	Lower	Higher

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.

Soln:-

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

Importance of Responsive Design in PWAs:

1] Better User Experience
2] Faster Load Time
3] SEO benefits
4] Cost effective.

Comparisons :-

Approach	How it works	Pros	Cons
Responsive	Use flexible grids and CSS media queries.	Works on all devices	Can be complex
Fluid	Use % based widths instead of fixed pixels	Work well on different screen size	Less control over

Key difference :-

1) Responive adapts dynamically to all screens
2) Fluid resizes smoothly but may not be fully optimized
3) Adaptive loads different layouts based on device type.

Q.3] Describes the lifecycle of services workers, including registration, installation and activation phases.

Soln :-

Lifecycle of services workers :-

A services 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.

e.g. :-

```
if ('serviceWorker' in navigator) {

    navigator.serviceWorker.register("/sw.js")
    .then(() => console.log("Service Registered"))

    .catch(error => console.log("Registered failed:",
                                error));

}
```

2] Installation phase.

i) The service worker downloads necessary files (HTML, CSS, JS) and stores them in cache.

ii) if successful, it moves to the activation phase.

Code eg:-

```
self.addEventListener('install', event => {

    event.waitUntil(
        caches.open('app-cache').then(cache => {

            return cache.addAll(['/', 'index.html',
                                 'styles.css']);
        })
    );
});
```

3] Activation phase.

→ The old services worker is replaced with new one.
→ unused cache files from the previous version are deleted

Final step : Fetch & sync.

Once activated, the services worker intercept network request, serves cached files and syncs data.

Q4] Explain the use of IndexedDB in the services workers for data storage.

Soln :-

Use of IndexedDB in service workers for Data Storage :-
IndexedDB is a Browser database that stores large amount of structured data like JSON objects.
It helps PWAs work offline by saving and retrieving data efficiency.

Why use IndexedDB in service workers?

1] offline support - Store data when offline and sync it later.

2] Efficient Storage - Saves structure data like user setting, 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 IndexedDB?

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('User');
let getUser = store.get(1);
getUser.onsuccess = function () {
    console.log (getUser result);
};
```