

EXPERIMENT NO. 1

MAD and PWA Lab

Aim: Installation and Configuration of Flutter Environment.

Theory:

Flutter is an open-source UI toolkit developed by Google that allows developers to create cross-platform applications using a single codebase. It simplifies the development process by providing a collection of pre-designed widgets, enabling seamless UI design for both Android and iOS platforms. Flutter utilizes the Dart programming language, which is optimized for building fast and scalable applications.

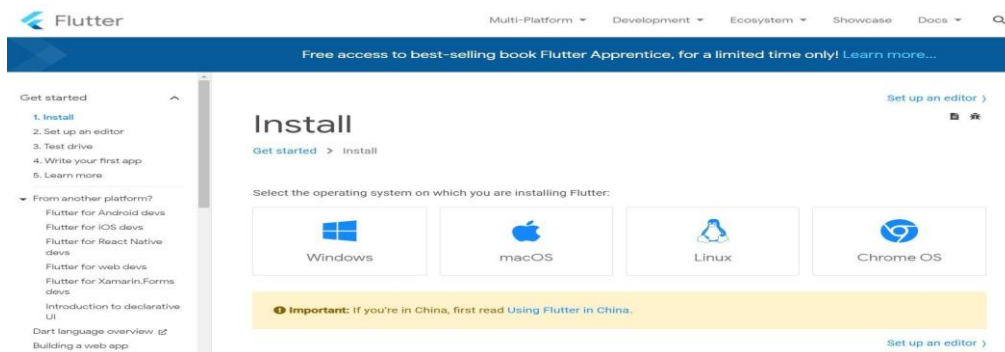
Flutter consists of two major components:

- **Software Development Kit (SDK):** A collection of essential tools for building applications, including a compiler that converts the code into native machine code for Android and iOS.
- **UI Framework (Widget Library):** A set of reusable UI components like buttons, text fields, and sliders that can be customized according to the app's requirements.

Step-by-Step Installation and Configuration:

1. Download Flutter SDK

- Visit the official Flutter website: [Flutter Install Guide](#).
- Click on the Windows icon to download the latest stable version of Flutter.



2.Extract and Set Up Flutter

- Once downloaded, extract the ZIP file and move the Flutter folder to the desired installation location, e.g., C:/Flutter.

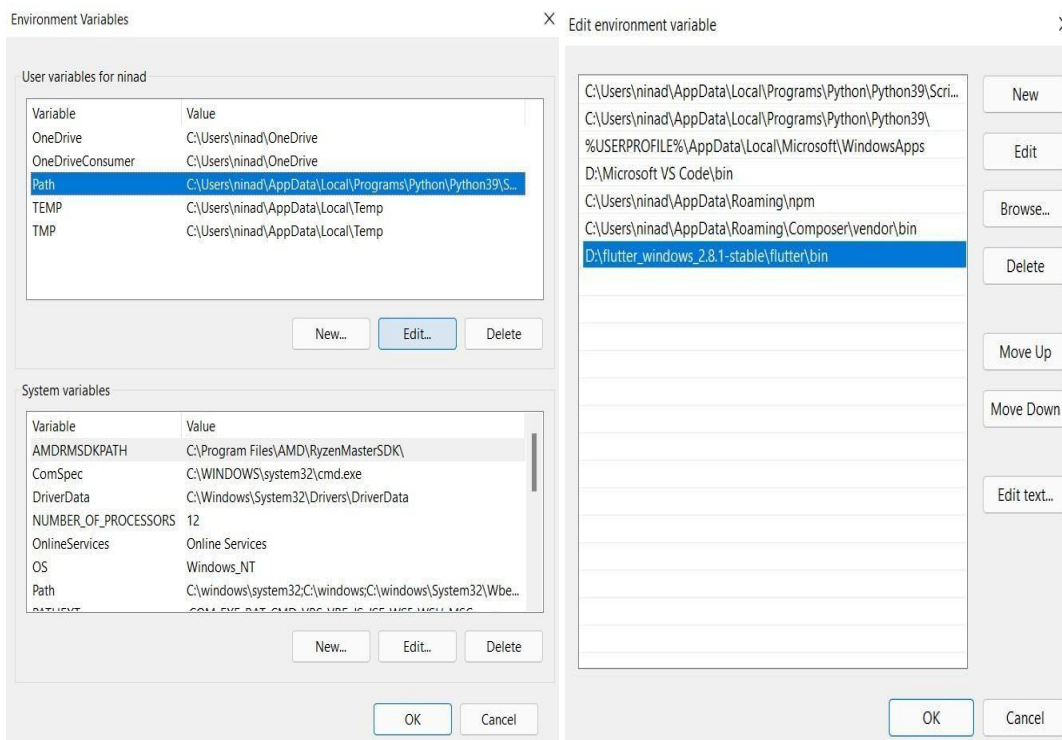
Update System Path

To use Flutter commands globally, add Flutter to the system path:

Open **My Computer > Properties > Advanced System Settings > Environment Variables**.

Locate the **Path** variable, click **Edit**, then **New**.

Add the path to the bin folder inside the Flutter directory. Click **OK** to save the changes.



- **Verify Flutter Installation**
Flutter command

Open the command prompt and enter:

```
C:\Users\HP>flutter
```

```
A new version of Flutter is available!
```

```
To update to the latest version, run "flutter upgrade".
```

```
Manage your Flutter app development.
```

```
Common commands:
```

```
flutter create <output directory>
```

```
Create a new Flutter project in the specified directory.
```

```
flutter run [options]
```

```
Run your Flutter application on an attached device or in an emulator.
```

flutter doctor

This command checks for system dependencies and lists any missing components.

Now, run the **\$ flutter doctor** command.

This command checks for all the requirements of Flutter app development and displays a report of the status of your Flutter installation.

```
C:\Users\HP>flutter doctor
```

```
Doctor summary (to see all details, run flutter doctor -v):
```

```
[✓] Flutter (Channel stable, 3.27.4, on Microsoft Windows [Version 10.0.26100.3624], locale en-IN)
```

```
[✓] Windows Version (Installed version of Windows is version 10 or higher)
```

```
[✓] Android toolchain - develop for Android devices (Android SDK version 35.0.1)
```

```
[✓] Chrome - develop for the web
```

```
[✓] Visual Studio - develop Windows apps (Visual Studio Build Tools 2019 16.11.39)
```

```
[✓] Android Studio (version 2024.2)
```

```
[✓] VS Code (version 1.98.2)
```

```
[✓] Connected device (3 available)
```

```
[✓] Network resources
```

```
• No issues found!
```

```
C:\Users\HP>
```

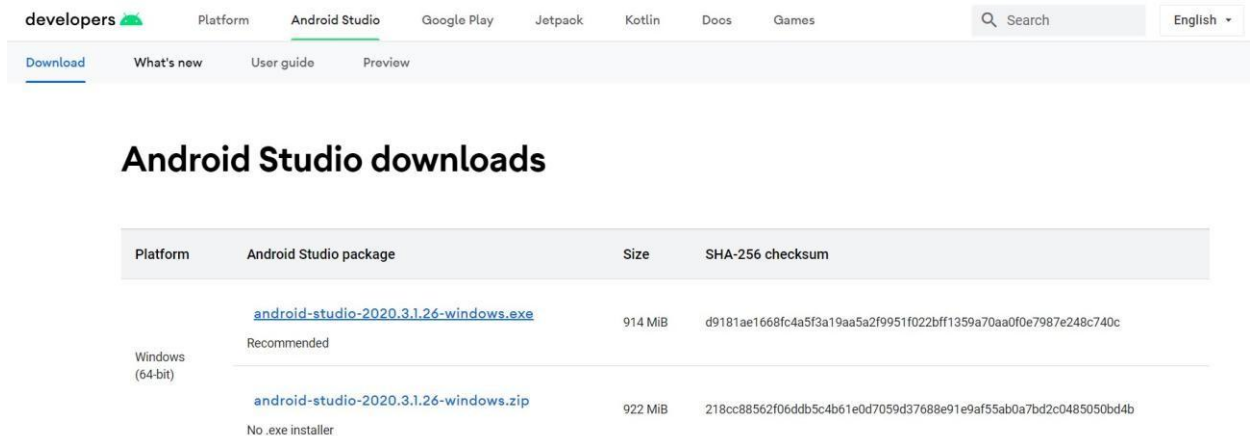
This command checks for system dependencies and lists any missing components.

Install Android Studio

- If flutter doctor detects a missing Android SDK, install **Android Studio**:

Open the command prompt and execute:

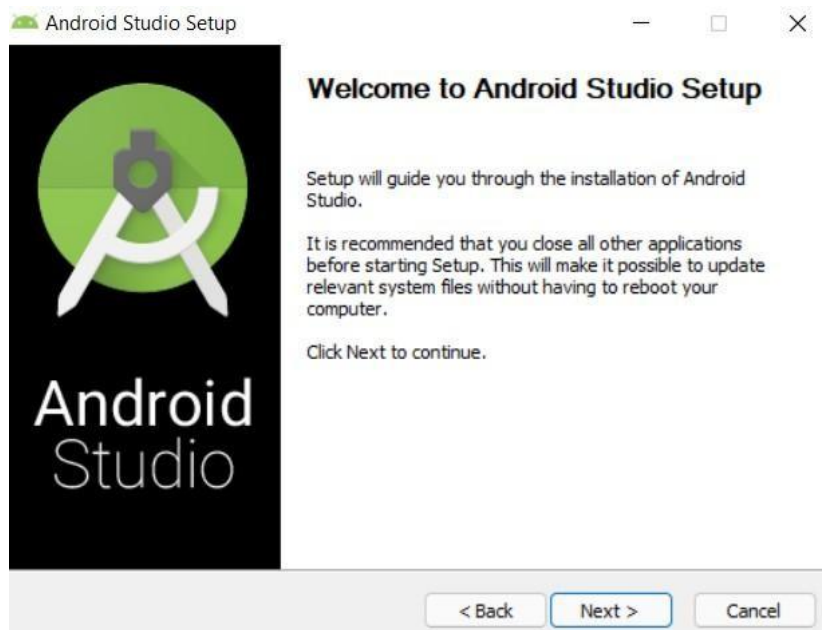
Download the latest version from the [official website](#).



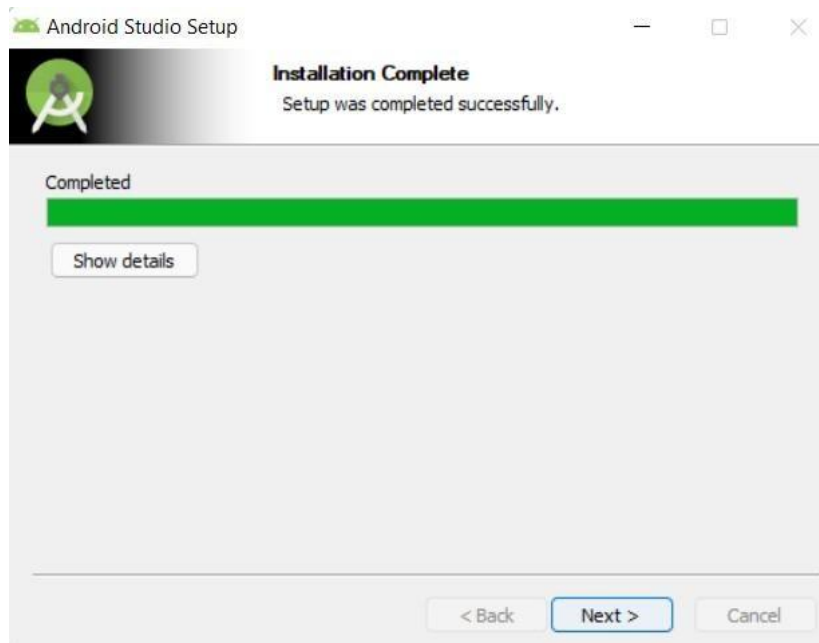
The screenshot shows the 'Android Studio downloads' page on the official website. It features a navigation bar with links like 'developers', 'Platform', 'Android Studio', 'Google Play', 'Jetpack', 'Kotlin', 'Doos', and 'Games'. Below the navigation bar, there's a section titled 'Android Studio downloads' with a table listing available packages for Windows (64-bit). The table has columns for 'Platform', 'Android Studio package', 'Size', and 'SHA-256 checksum'. Two packages are listed: 'Recommended' (914 MiB) and 'No .exe installer' (922 MiB).

Platform	Android Studio package	Size	SHA-256 checksum
Windows (64-bit)	android-studio-2020.3.1.26-windows.exe Recommended	914 MiB	d9181ae1668fc4a5f3a19aa5a2f9951f022bffa1359a70aa0f0e7987e248c740c
	android-studio-2020.3.1.26-windows.zip No .exe installer	922 MiB	218cc88562f06ddb5c4b61e0d7059d37688e91e9af55ab0a7bd2c0485050bd4b

Run the installation wizard and follow the on-screen instructions.

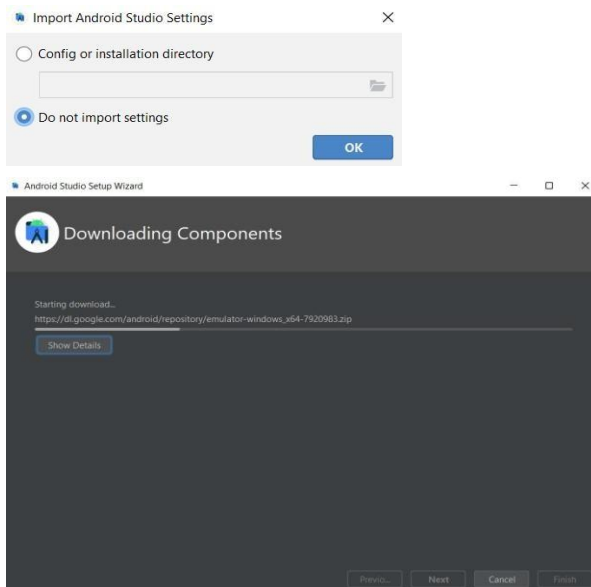


Follow the steps of the installation wizard. Once the installation wizard completes, you will get the following screen.



- a. 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.

Choose '**Don't import Settings**' when prompted, then click **OK**.



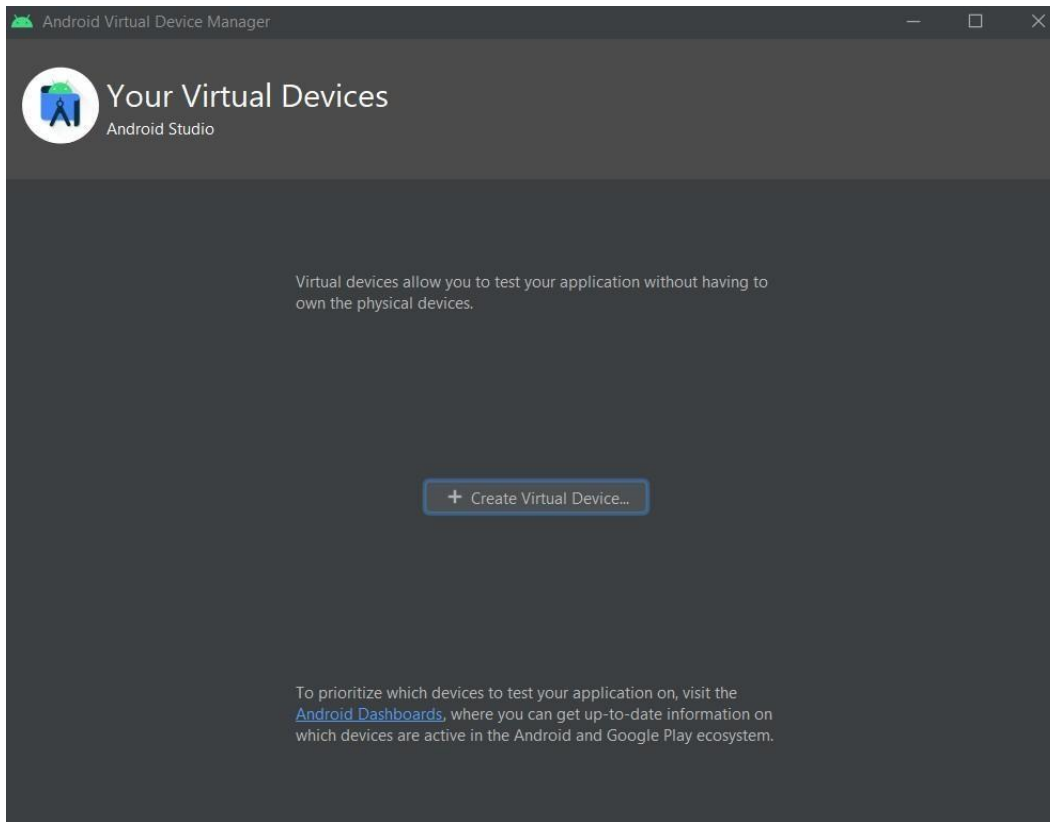
- b. Run the **\$ flutter doctor** command and Run **flutter doctor --android-licenses** command.

```
C:\Users\HP>flutter doctor
Doctor summary (to see all details, run flutter doctor -v):
[✓] Flutter (Channel stable, 3.27.4, on Microsoft Windows [Version 10.0.26100.3624], locale en-IN)
[✓] Windows Version (Installed version of Windows is version 10 or higher)
[✓] Android toolchain - develop for Android devices (Android SDK version 35.0.1)
[✓] Chrome - develop for the web
[✓] Visual Studio - develop Windows apps (Visual Studio Build Tools 2019 16.11.39)
[✓] Android Studio (version 2024.2)
[✓] VS Code (version 1.98.2)
[✓] Connected device (3 available)
[✓] Network resources

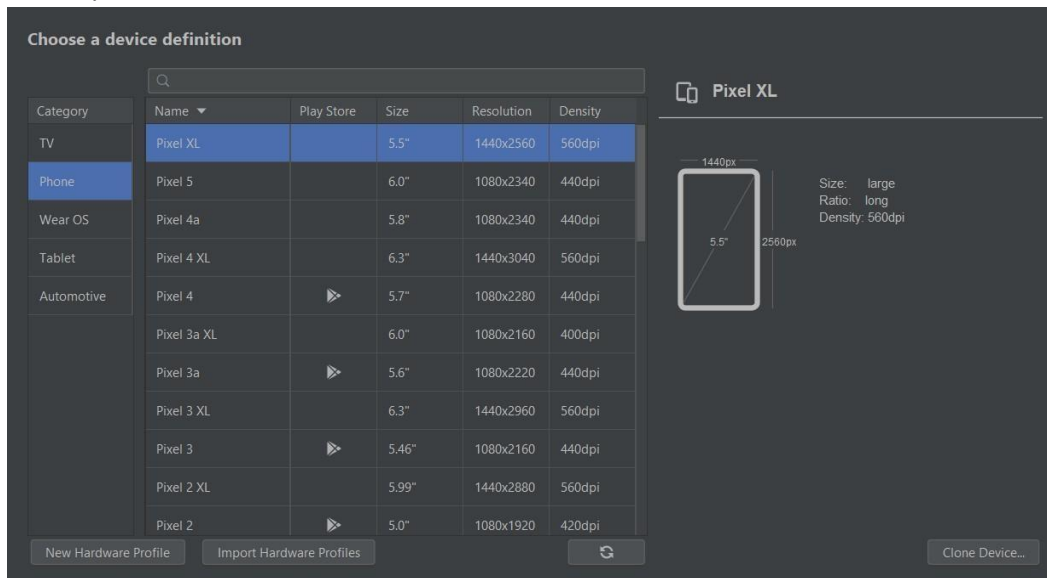
• No issues found!

C:\Users\HP>
```

1. Next, you need to set up an Android emulator. It is responsible for running and testing the Flutter application.
- a. 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.

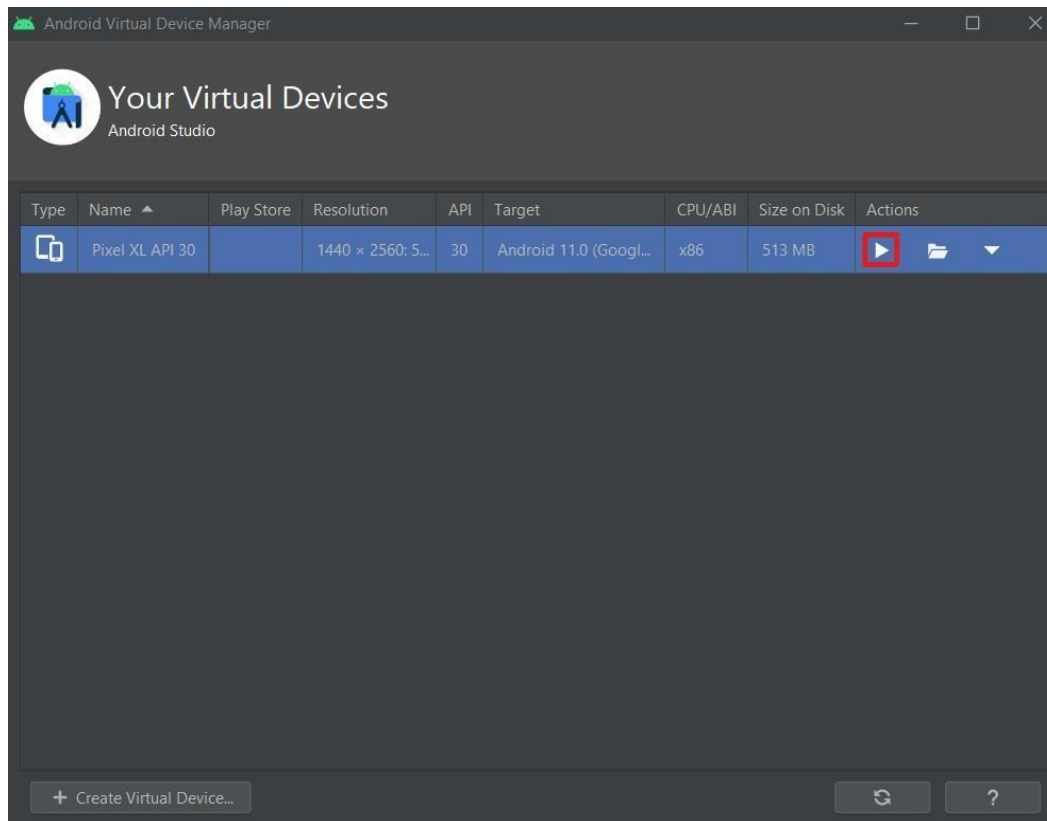


- b. Choose your device definition and click on **Next**.

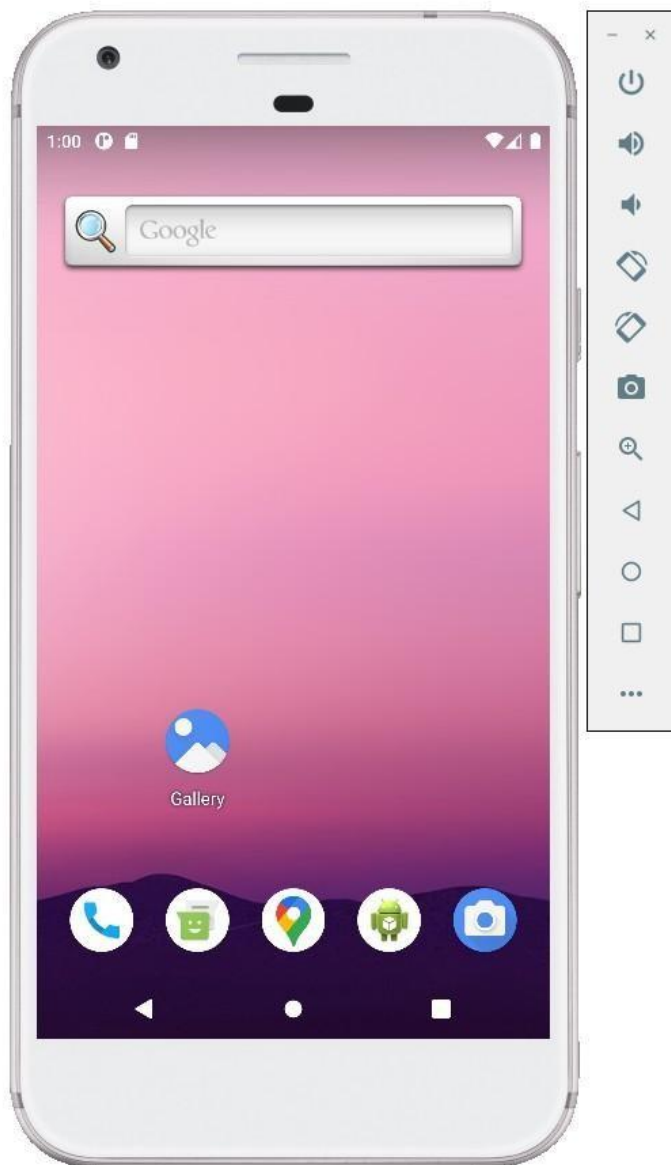


- c. Select the system image for the latest Android version and click on **Next**.

- d. Now, verify the all AVD configuration. If it is correct, click on **Finish**. The following screen appears.



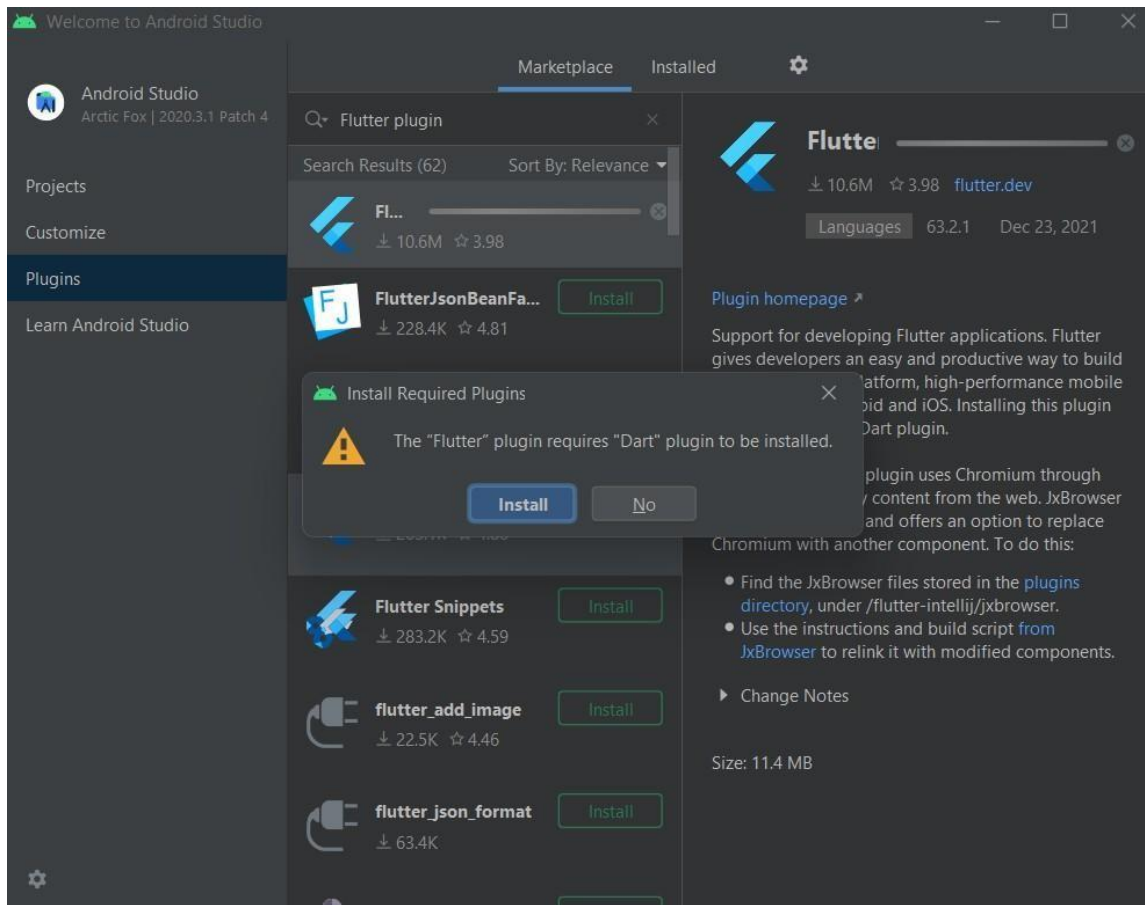
- e. Last, click on the icon pointed into the **red color rectangle**. The Android emulator displayed as shown below screen.



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

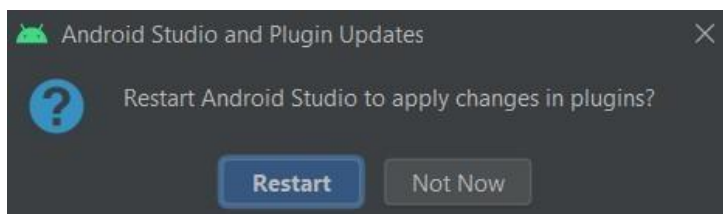
- a. Open the Android Studio and then go to **File -> Settings -> Plugins**.

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



c.

Restart the Android Studio.



Conclusion: Hence, we understood how to install and configure the Flutter environment by installing the Flutter SDK, installing and setting up Android Studio and in the end creating and adding a virtual device to the Android Studio.

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

Theory:

Flutter is a popular open-source UI toolkit developed by Google that allows developers to create beautiful, natively compiled applications for mobile, web, and desktop from a single codebase. The heart of Flutter lies in its widget system. Everything in Flutter is a widget—from the entire app layout to individual components such as buttons or text.

In the context of a Quiz App, using Flutter widgets effectively allows developers to design smooth, interactive, and visually appealing interfaces that enhance the user experience. Below are the common widgets used in Flutter and how they are utilized in a Quiz App:

1. **Scaffold**

Scaffold is the basic structure that provides the framework for implementing material design layout in the app. It supports components like AppBar, Drawer, BottomNavigationBar, and FloatingActionButton. In a quiz app, it holds the overall layout for each screen.

2. **AppBar**

AppBar is a widget provided by Scaffold that appears at the top of the screen. It is generally used to show the title of the screen, navigation icons, or action buttons. In a quiz app, it usually displays the quiz title or the current question number.

3. **Text**

The Text widget is used to display strings of text. In a quiz app, it is commonly used to show questions, instructions, scores, and feedback messages like “Correct Answer” or “Try Again.”

4. **Column and Row**

Column arranges child widgets vertically, while Row arranges them horizontally. They are the foundation for building structured layouts in Flutter. For example, options in a quiz can be displayed in a Column, and control buttons like “Next” and “Previous” can be placed in a Row.

5. **Container and Card**

Container is a versatile widget that can contain any other widget and apply padding, margin, alignment, background color, or decoration. Card is a special type of Container with elevation and rounded corners, making it suitable for displaying quiz options in a clean, elevated block.

6. **Buttons (ElevatedButton, TextButton, etc.)**

Buttons are essential for interaction. ElevatedButton is used for primary actions like submitting an answer or moving to the next question. TextButton is used for secondary actions like “Retry” or “Back.”

7. **ListTile**

ListTile provides a simple way to create tappable options with a title, subtitle, leading or trailing icons. In a quiz app, it is ideal for displaying each option that the user can select.

8. Image

The Image widget is used to show images, which can be loaded from local assets or the internet. It can be used for branding, question images, or icons in the quiz.

9. AlertDialog

This widget is used to show a popup dialog with a message, title, and action buttons. It is helpful in showing the result of the quiz, confirmation to exit the quiz, or wrong answer feedback.

10. LinearProgressIndicator

This is a horizontal progress bar that visually indicates progress. In a quiz app, it is used to show how many questions have been attempted or completed.

11. Navigator

Navigator is used for screen navigation. It allows moving from one screen to another, such as from the home screen to the quiz screen, or from the quiz screen to the result screen.

- By combining these widgets, developers can design a responsive and well-structured quiz interface with clean layout, user interaction, and real-time feedback.
- Scaffold is used to build the main structure of the app. It holds everything including the AppBar and the body of the screen. AppBar displays the title of the quiz and helps users understand where they are in the app.
- Text widget displays the actual quiz content, like questions, options, or scores. Column helps organize the question followed by multiple options vertically. Row is used when two widgets need to be placed side-by-side, such as "Next" and "Submit" buttons.
- Container and Card are used to design the layout with styling, padding, margins, background color, and elevation. Card is often used to display each quiz option nicely separated from others.
- ElevatedButton is commonly used for interaction when users want to submit their answer or go to the next question. TextButton is useful for less prominent actions like "Skip" or "Retry."
- ListTile is a simple but powerful widget to display one quiz option. It has a built-in touch handler, so when a user taps an option, you can easily process it.
- Image is used to make the app visually appealing, such as placing the app logo on the home page or showing an image-based question.
- AlertDialog is shown after an answer is selected to give feedback or to confirm actions such as exiting the quiz or completing the quiz.
- LinearProgressIndicator shows how far along the user is in the quiz. For example, if the quiz has 10 questions, you can use this widget to indicate if the user has completed 3 out of 10.
- Navigator is used to move between different screens of the quiz app. From the home screen, it can navigate to the quiz screen, and then to the result screen after completion.

These widgets collectively help to build an interactive and functional quiz app with clean navigation and smooth user experience.

```
import 'dart:math';

import 'package:cloud_firestore/cloud_firestore.dart';
import 'package:firebase_auth/firebase_auth.dart';
import 'package:flutter/material.dart';
import 'leaderboard_page.dart';
// import 'signin_page.dart';
import 'login.dart';
import 'quiz_page.dart';
import 'profile_page.dart';

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

  @override
  State<HomePage> createState() => _HomePageState();
}

class _HomePageState extends State<HomePage> {
  String username = "User";
  String profilePicUrl =
    "https://avatar.iran.liara.run/public"; // Random avatar placeholder
  final FirebaseAuth _auth = FirebaseAuth.instance;
  int _selectedIndex = 0;

  final List<Color> categoryColors = [
    Colors.greenAccent.shade100,
```

```
Colors.blueAccent.shade100,  
Colors.purpleAccent.shade100,  
Colors.orangeAccent.shade100,  
Colors.redAccent.shade100,  
];
```

```
final List<IconData> categoryIcons = [  
    Icons.menu_book,  
    Icons.tv,  
    Icons.fastfood,  
    Icons.science,  
    Icons.account_balance,  
    Icons.category,  
    Icons.sports_esports,  
    Icons.music_note,  
    Icons.flight,  
    Icons.computer,  
];
```

```
@override  
void initState() {  
    super.initState();  
    fetchUserData();  
}
```

```
Future<void> fetchUserData() async {
```

```
User? user = _auth.currentUser;

if (user != null) {

  DocumentSnapshot userDoc = await FirebaseFirestore.instance
    .collection('users')
    .doc(user.uid)
    .get();

  if (userDoc.exists) {

    setState(() {

      username = userDoc['username'];

      int randomNumber =

        Random().nextInt(50) + 1; // Generate a number between 1-50

      profilePicUrl =

        "https://avatar iran.liara.run/public/$randomNumber"; // Random avatar

    });

  }

}

}

Stream<QuerySnapshot> fetchCategories() {

  return FirebaseFirestore.instance.collection('categories').snapshots();

}

void _onItemTapped(int index) {

  if (index == _selectedIndex) return;

  setState(() {
```

```
    _selectedIndex = index;  
  });
```

```
User? user = _auth.currentUser; // Get the current user
```

```
switch (index) {  
  case 0:  
    Navigator.pushReplacement(  
      context,  
      MaterialPageRoute(builder: (context) => const HomePage()),  
    );  
    break;  
  case 1:  
    Navigator.pushReplacement(  
      context,  
      MaterialPageRoute(  
        builder: (context) => const LeaderboardPage(category: 'Science')),  
    );  
    break;  
  case 2:  
    if (user != null) {  
      Navigator.pushReplacement(  
        context,  
        MaterialPageRoute(  
          builder: (context) =>  
            ProfilePage(userId: user.uid), // Pass user ID
```

```
    );  
  }  
  break;  
}  
}
```

@override

```
Widget build(BuildContext context) {  
  return Scaffold(  
    backgroundColor: const Color.fromARGB(255, 237, 237, 241),  
    appBar: AppBar(  
      title: const Text(  
        "QUIZZIFY",  
        style: TextStyle(  
          fontSize: 22, fontWeight: FontWeight.bold, color: Colors.white),  
        ),  
      centerTitle: true,  
      backgroundColor: Colors.deepPurple,  
      actions: [  
        IconButton(  
          icon: const Icon(Icons.logout, color: Colors.white),  
          onPressed: () async {  
            await _auth.signOut();  
            Navigator.pushReplacement(  
              context,  
              MaterialPageRoute(builder: (context) => const SignInPage()),  
            );  
          },  
        ),  
      ],  
    ),  
  );  
}
```



```
);
},
),
],
),
body: SafeArea(
  child: Padding(
    padding: const EdgeInsets.all(16.0),
    child: Column(
      crossAxisAlignment: CrossAxisAlignment.start,
      children: [
        // Good Morning Section with Background Box
        Container(
          padding: const EdgeInsets.all(20),
          decoration: BoxDecoration(
            color: Colors.deepPurple.shade100, // Background color
            borderRadius: BorderRadius.circular(15), // Rounded corners
          ),
          child: Row(
            mainAxisAlignment: MainAxisAlignment.spaceBetween,
            children: [
              Column(
                crossAxisAlignment: CrossAxisAlignment.start,
                children: [
                  const Text(
                    "GOOD MORNING",
```

```
        style: TextStyle(
          fontSize: 22, // Increased font size
          fontWeight: FontWeight.bold,
          color: Colors.deepPurple),
      ),
      Text(
        username,
        style: const TextStyle(
          fontSize: 34, // Increased font size
          fontWeight: FontWeight.bold,
          color: Colors.black),
      ),
    ],
  ),
  // Profile Picture
  CircleAvatar(
    radius: 40,
    backgroundImage: NetworkImage(profilePicUrl),
    backgroundColor: Colors.grey.shade300,
  ),
],
),
),
const SizedBox(height: 20),

// Quiz Categories Section
```

```
const Text(  
  "Quiz Categories",  
  style: TextStyle(  
    fontSize: 22,  
    fontWeight: FontWeight.bold,  
    color: Colors.black),  
),  
const SizedBox(height: 12),  
  
Expanded(  
  child: StreamBuilder<QuerySnapshot>(  
    stream: fetchCategories(),  
    builder: (context, snapshot) {  
      if (!snapshot.hasData) {  
        return const Center(child: CircularProgressIndicator());  
      }  
      var categories = snapshot.data!.docs;  
      return Scrollbar(  
        thickness: 6,  
        radius: const Radius.circular(10),  
        child: SingleChildScrollView(  
          child: Column(  
            children: List.generate(categories.length, (index) {  
              var data = categories[index].data()  
              as Map<String, dynamic>;  
              Color boxColor =
```

```
        categoryColors[index % categoryColors.length];

IconData icon =

        categoryIcons[index % categoryIcons.length];

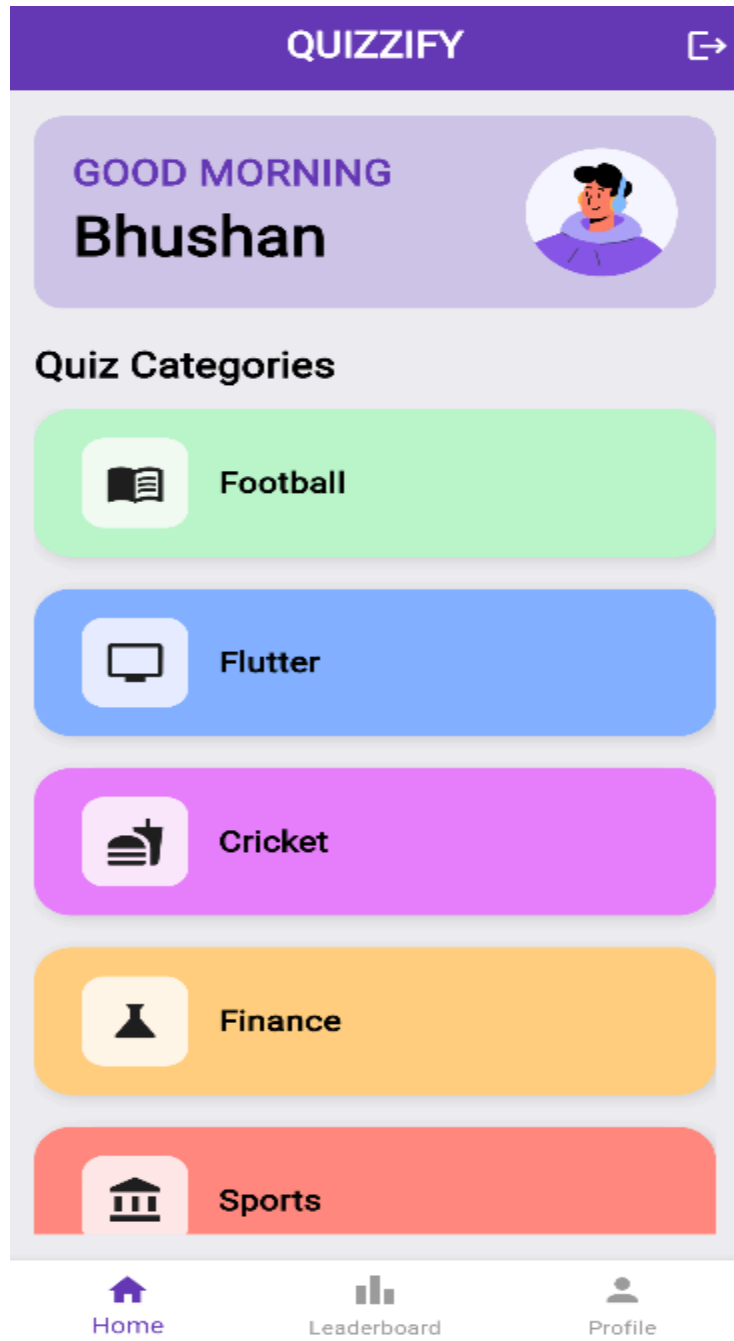
return GestureDetector(
  onTap: () {
    Navigator.push(
      context,
      MaterialPageRoute(
        builder: (context) =>
          QuizPage(category: data['name']),
      ),
    );
  },
  child: Container(
    margin: const EdgeInsets.only(bottom: 20),
    padding: const EdgeInsets.symmetric(
      vertical: 18, horizontal: 25),
    decoration: BoxDecoration(
      color: boxColor,
      borderRadius: BorderRadius.circular(20),
      boxShadow: [
        BoxShadow(
          color: Colors.black12,
          blurRadius: 6,
          offset: Offset(2, 2)),
```

```
    ],  
  ),  
  child: Row(  
    children: [  
      Container(  
        padding: const EdgeInsets.all(12),  
        decoration: BoxDecoration(  
          color: Colors.white.withOpacity(0.8),  
          borderRadius: BorderRadius.circular(12),  
        ),  
        child: Icon(  
          icon,  
          color: Colors.black87,  
          size: 32,  
        ),  
      ),  
      const SizedBox(width: 16),  
      Expanded(  
        child: Text(  
          data['name'],  
          style: const TextStyle(  
            fontSize: 18,  
            fontWeight: FontWeight.bold,  
            color: Colors.black),  
        ),  
      ),  
    ],  
  ),  
),
```

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

// Bottom Navigation Bar
bottomNavigationBar: BottomNavigationBar(
  currentIndex: _selectedIndex,
  onTap: _onItemTapped,
  selectedItemColor: Colors.deepPurple,
  unselectedItemColor: Colors.grey,
  backgroundColor: Colors.white,
  elevation: 10,
  showUnselectedLabels: true,
  items: const [
```

```
        BottomNavigationBarItem(icon: Icon(Icons.home), label: "Home"),  
        BottomNavigationBarItem(  
            icon: Icon(Icons.leaderboard), label: "Leaderboard"),  
        BottomNavigationBarItem(icon: Icon(Icons.person), label: "Profile"),  
    ],  
),  
);  
}  
}
```

**Conclusion:**

Using common widgets helps in building a responsive, maintainable, and scalable Flutter application. In my Quiz App, these widgets form the core building blocks of the UI. By understanding how to combine and customize them, you ensure a smooth user experience and professional-looking design.

Experiment 03 : To include icons, images, fonts in Flutter app**Theory:**

A visually appealing and user-friendly interface is key to enhancing the user experience (UX) in Flutter applications. In your Quiz App's SignUpPage, you've effectively used icons, custom images, and fonts to create an engaging onboarding experience. Here's how each plays a role:

1. Icons

- Purpose: Icons provide visual cues, improve navigation, and enhance form usability.
- Used in input fields for username, email, password, and role to indicate the type of input required.
- Widget Used: Icon, Icons.person, Icons.email, Icons.lock, etc

2. Images

- Purpose: Images (like logos or banners) reinforce branding and break monotony in the UI.
- Displays the Quizzify logo at the top of the sign-up screen.

3. Fonts

- Purpose: Custom fonts give your app a unique visual identity and improve readability.
- Aligns typography with the app's theme.
- Helps differentiate important texts (like headings and labels).

```
import 'package:firebase_auth/firebase_auth.dart';
import 'package:flutter/gestures.dart';
import 'package:flutter/material.dart';
import 'package:cloud_firestore/cloud_firestore.dart';
import 'homepage.dart';
import 'signuppage.dart';
import 'admin.dart';
import 'forgot_page.dart';
```

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

  @override
  State<SignInPage> createState() => _SignInPageState();
}
```

```
class _SignInPageState extends State<SignInPage> {
  final FirebaseAuth _auth = FirebaseAuth.instance;
  final TextEditingController _emailController = TextEditingController();
  final TextEditingController _passwordController = TextEditingController();
  final _formKey = GlobalKey<FormState>();
  bool _isLoading = false;

  Future<void> _signIn() async {
    if (!_formKey.currentState!.validate()) {
      return; // Don't proceed if form validation fails
    }

    setState(() {
      _isLoading = true;
    });

    try {
      UserCredential userCredential = await _auth.signInWithEmailAndPassword(
        email: _emailController.text.trim(),
        password: _passwordController.text.trim(),
      );

      DocumentSnapshot userDoc = await FirebaseFirestore.instance
        .collection('users')
        .doc(userCredential.user!.uid)
        .get();

      if (userDoc.exists) {
        String role = userDoc['role'];

        if (role == 'admin') {
          Navigator.pushReplacement(
            context,
            MaterialPageRoute(builder: (context) => const AdminPage()),
          );
        } else {
          Navigator.pushReplacement(
            context,
            MaterialPageRoute(builder: (context) => const HomePage()),
          );
        }
      }
    } catch (e) {
      // Handle error
    }
  }
}
```

```
        );
    }
}
} on FirebaseAuthException catch (e) {
    _showErrorDialog(e);
} catch (e) {
    _showErrorDialog(e);
}

setState(() {
    _isLoading = false;
});
}

// Show Alert Dialog for Errors
void _showErrorDialog(dynamic error) {
    String errorMessage = "An error occurred. Please try again.";

    if (error is FirebaseAuthException) {
        switch (error.code) {
            case "invalid-email":
                errorMessage = "Invalid email format.";
                break;
            case "user-not-found":
                errorMessage = "No account found with this email.";
                break;
            case "wrong-password":
                errorMessage = "Incorrect password. Please try again.";
                break;
            case "user-disabled":
                errorMessage = "This account has been disabled.";
                break;
            case "too-many-requests":
                errorMessage = "Too many failed attempts. Please try again later.";
                break;
            default:
                errorMessage = "Login failed: ${error.message}";
        }
    } else {
        errorMessage = "Unexpected error: ${error.toString()}";
    }
}
```

```
}

showDialog(
  context: context,
  builder: (context) => AlertDialog(
    title: const Text("Login Error"),
    content: Text(errorMessage),
    actions: [
      TextButton(
        onPressed: () => Navigator.pop(context),
        child: const Text("OK"),
      ),
    ],
  ),
);
}
```

@override

Widget build(BuildContext context) {

return Scaffold(

body: Container(

decoration: const BoxDecoration(

gradient: LinearGradient(

colors: [Color(0xFF4A00E0), Color(0xFF8E2DE2)],

begin: Alignment.topCenter,

end: Alignment.bottomCenter,

),

),

child: Center(

child: Padding(

padding: const EdgeInsets.symmetric(horizontal: 24),

child: Form(

key: _formKey,

child: Column(

mainAxisAlignment: MainAxisAlignment.min,

children: [

// App Icon

Image.asset(

"assets/images/quizzify_logo.png", // Ensure this image is in the assets

folder

```
        height: 100,
      ),
      const SizedBox(height: 10),

      // App Name
      const Text(
        "QUIZZIFY",
        style: TextStyle(
          fontSize: 32,
          fontWeight: FontWeight.bold,
          color: Colors.white,
        ),
      ),
      const SizedBox(height: 20),

      const Text(
        "Welcome Back",
        style: TextStyle(
          fontSize: 28,
          fontWeight: FontWeight.bold,
          color: Colors.white,
        ),
      ),
      const SizedBox(height: 5),
      const Text(
        "Enter your credentials to login",
        style: TextStyle(fontSize: 16, color: Colors.white70),
      ),
      const SizedBox(height: 30),

      // Email Field
      TextFormField(
        controller: _emailController,
        keyboardType: TextInputType.emailAddress,
        decoration: InputDecoration(
          labelText: "Email",
          labelStyle:
            const TextStyle(color: Colors.white70, fontSize: 16),
          filled: true,
          fillColor: Colors.white.withOpacity(0.2),
```

```
    prefixIcon: const Icon(Icons.email, color: Colors.white),
    border: OutlineInputBorder(
      borderRadius: BorderRadius.circular(12),
      borderSide: BorderSide.none,
    ),
  ),
  style: const TextStyle(color: Colors.white, fontSize: 18),
  validator: (value) {
    if (value == null || value.isEmpty) {
      return "Please enter your email";
    } else if (!value.contains('@') || !value.contains('.')) {
      return "Enter a valid email";
    }
    return null;
  },
),
```

```
const SizedBox(height: 15),
```

```
// Password Field
```

```
TextFormField(
  controller: _passwordController,
  obscureText: true,
  decoration: InputDecoration(
    labelText: "Password",
    labelStyle:
      const TextStyle(color: Colors.white70, fontSize: 16),
    filled: true,
    fillColor: Colors.white.withOpacity(0.2),
    prefixIcon: const Icon(Icons.lock, color: Colors.white),
    border: OutlineInputBorder(
      borderRadius: BorderRadius.circular(12),
      borderSide: BorderSide.none,
    ),
  ),
  style: const TextStyle(color: Colors.white, fontSize: 18),
  validator: (value) {
    if (value == null || value.isEmpty) {
      return "Please enter your password";
    } else if (value.length < 4 || value.length > 8) {
```

```
        return "Password must be 4-8 characters";
      }
      return null;
    },
  ),

  const SizedBox(height: 20),

  // Login Button
  SizedBox(
    width: double.infinity,
    height: 50,
    child: ElevatedButton(
      onPressed: _isLoading ? null : _signIn,
      style: ElevatedButton.styleFrom(
        backgroundColor: Colors.white,
        shape: RoundedRectangleBorder(
          borderRadius: BorderRadius.circular(12),
        ),
      ),
    ),
    child: _isLoading
      ? const CircularProgressIndicator(
          valueColor: AlwaysStoppedAnimation<Color>(
            Colors.deepPurple),
        )
      : const Text(
          "Login",
          style: TextStyle(
            color: Colors.deepPurple,
            fontSize: 16,
            fontWeight: FontWeight.bold,
          ),
        ),
  ),

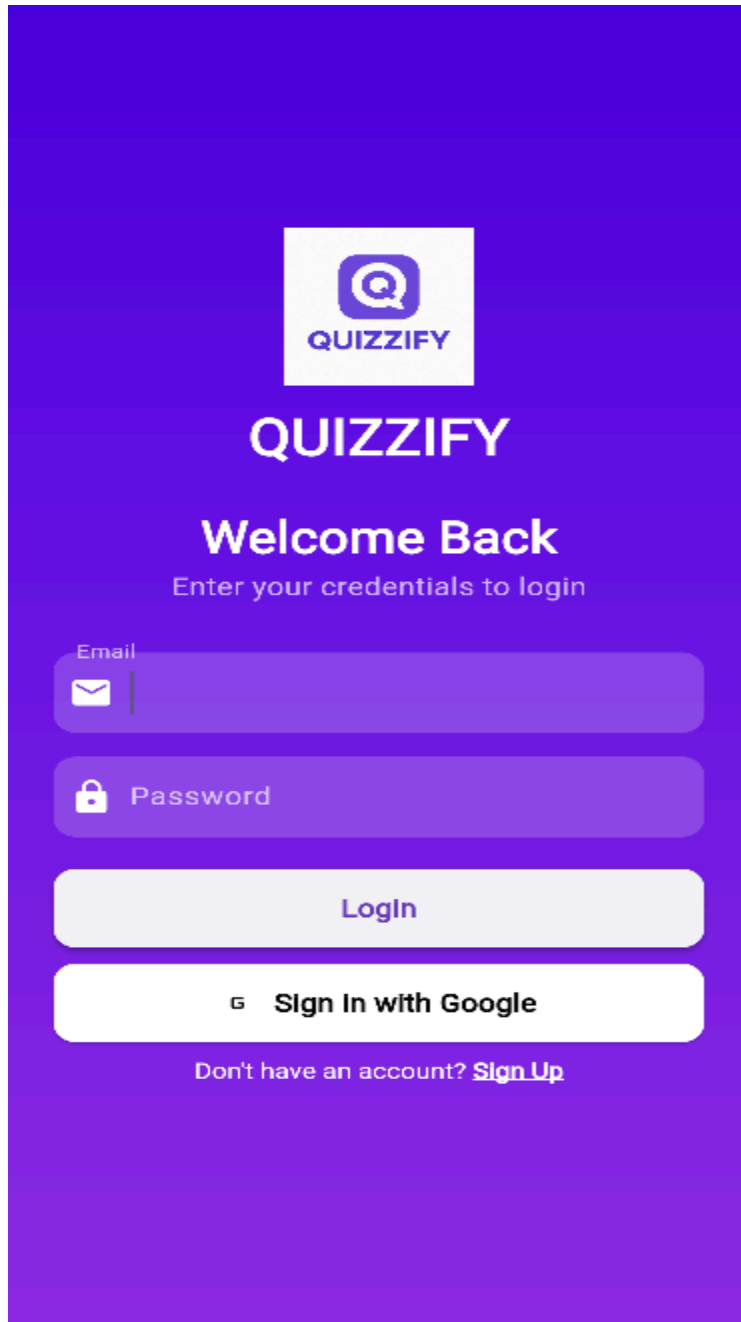
  const SizedBox(height: 15),

  // Forgot Password
  Align(
    alignment: Alignment.centerRight,
```

```
child: TextButton(  
  onPressed: () {  
    Navigator.push(  
      context,  
      MaterialPageRoute(  
        builder: (context) => ForgotPasswordPage()),  
    );  
  },  
child: const Text(  
  "Forgot Password?",  
  style: TextStyle(  
    color: Colors.white,  
    decoration: TextDecoration.underline,  
  ),  
,  
,  
,  
,  
,  
  
// Sign-Up Navigation  
RichText(  
  text: TextSpan(  
    style: const TextStyle(fontSize: 14, color: Colors.white),  
    children: [  
      const TextSpan(text: "Don't have an account? "),  
      TextSpan(  
        text: "Sign Up",  
        style: const TextStyle(  
          color: Colors.white,  
          fontWeight: FontWeight.bold,  
          decoration: TextDecoration.underline,  
        ),  
      ),  
      recognizer: TapGestureRecognizer()  
        ..onTap = () {  
          Navigator.push(  
            context,  
            MaterialPageRoute(  
              builder: (context) => const SignUpPage()),  
          );  
        },  
    ),  
  ),  
)
```



```
    ],  
  ),  
),  
const SizedBox(height: 20),  
],  
),  
),  
),  
),  
),  
);  
}  
}
```

**Conclusion:**

In the SignUpPage of your Quiz App, you've effectively demonstrated the use of icons, images, and fonts—three key UI elements that elevate the look and feel of a Flutter app.

Icons provide intuitive form navigation.

Images add branding and engagement.

Fonts (default or custom) improve visual hierarchy and tone.

Experiment 04: To create an interactive Form using form widget**Theory:**

In Flutter, forms are a critical part of collecting user input. They allow validation, state tracking, and control over multiple form fields. Your SignUpPage is an excellent example of building a fully functional and interactive Form.

1. Form Widget

- Purpose: Acts as a container for grouping form fields and handling their validation.

2. TextFormField Widget

- Purpose: Used for user input like username, email, and password.
- Each field includes:
 - A controller to access input values.
 - A validator to define validation rules.

3. DropdownButtonFormField

- Used to allow the user to select a role (student or admin).

4. Form Validation & Submission

- Each input is validated based on its rules (validator) before proceeding with Firebase Auth registration.

```
import 'package:firebase_auth/firebase_auth.dart';
import 'package:flutter/material.dart';
import 'package:cloud_firestore/cloud_firestore.dart';
import 'homepage.dart';
import 'signin_page.dart';
import 'admin.dart';
```

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

  @override
  State<SignUpPage> createState() => _SignUpPageState();
}
```

```
class _SignUpPageState extends State<SignUpPage> {
```

```
final FirebaseAuth _auth = FirebaseAuth.instance;
final FirebaseFirestore _firestore = FirebaseFirestore.instance;

final TextEditingController _usernameController = TextEditingController();
final TextEditingController _emailController = TextEditingController();
final TextEditingController _passwordController = TextEditingController();
final _formKey = GlobalKey<FormState>();
bool _isLoading = false;

String _selectedRole = 'student'; // Default role

Future<void> _signUp() async {
  if (!_formKey.currentState!.validate()) return;

  setState(() => _isLoading = true);

  try {
    // Create user
    UserCredential userCredential =
      await _auth.createUserWithEmailAndPassword(
        email: _emailController.text.trim(),
        password: _passwordController.text.trim(),
      );

    User? user = userCredential.user;

    if (user != null) {
      // Send email verification
      await user.sendEmailVerification();

      // Store user data in Firestore
      await _firestore.collection('users').doc(user.uid).set({
        'username': _usernameController.text.trim(),
        'email': _emailController.text.trim(),
        'role': _selectedRole,
        'createdAt': Timestamp.now(),
      });

      // Show verification message
      _showEmailVerificationDialog();
    }
  }
```

```
} on FirebaseAuthException catch (e) {
  ScaffoldMessenger.of(context).showSnackBar(
    SnackBar(content: Text("Error: ${e.message}")),
  );
}

setState(() => _isLoading = false);
}

void _showEmailVerificationDialog() {
  showDialog(
    context: context,
    builder: (context) => AlertDialog(
      title: const Text("Verify Your Email"),
      content: const Text(
        "A verification email has been sent to your email address. Please verify your email before logging in."),
      actions: [
        TextButton(
          onPressed: () {
            Navigator.pushReplacement(
              context,
              MaterialPageRoute(builder: (context) => const SignInPage()),
            );
          },
          child: const Text("OK"),
        ),
      ],
    ),
  );
}

@override
Widget build(BuildContext context) {
  return Scaffold(
    body: Container(
      width: double.infinity,
      padding: const EdgeInsets.all(24.0),
      decoration: const BoxDecoration(
        gradient: LinearGradient(
          colors: [Color(0xFF4A00E0), Color(0xFF8E2DE2)],

```

```
begin: Alignment.topCenter,
end: Alignment.bottomCenter,
),
),
child: Center(
  child: SingleChildScrollView(
    child: Form(
      key: _formKey,
      child: Column(
        crossAxisAlignment: CrossAxisAlignment.center,
        children: [
          // Quizzify Logo
          Image.asset("assets/images/quizzify_logo.png", height: 100),
          const SizedBox(height: 10),

          // App Name
          const Text(
            "QUIZZIFY",
            style: TextStyle(
              fontSize: 32,
              fontWeight: FontWeight.bold,
              color: Colors.white),
          ),
          const SizedBox(height: 20),

          const Text("Sign Up",
            style: TextStyle(
              fontSize: 28,
              fontWeight: FontWeight.bold,
              color: Colors.white)),
          const SizedBox(height: 8),
          const Text("Create your account",
            style: TextStyle(color: Colors.white70, fontSize: 16)),
          const SizedBox(height: 24),

          // Username Field
          TextFormField(
            controller: _usernameController,
            decoration: _buildInputDecoration("Username", Icons.person),
            style: const TextStyle(color: Colors.white, fontSize: 14),
            validator: (value) =>
```

```
        value!.isEmpty ? "Please enter a username" : null,
    ),
    const SizedBox(height: 16),

    // Email Field
    TextFormField(
      controller: _emailController,
      decoration: _buildInputDecoration("Email", Icons.email),
      keyboardType: TextInputType.emailAddress,
      style: const TextStyle(color: Colors.white, fontSize: 16),
      validator: (value) {
        if (value == null || value.isEmpty)
          return "Please enter your email";
        if (!value.contains("@") || !value.contains("."))
          return "Enter a valid email";
        return null;
      },
    ),
    const SizedBox(height: 16),

    // Password Field
    TextFormField(
      controller: _passwordController,
      decoration: _buildInputDecoration("Password", Icons.lock),
      obscureText: true,
      style: const TextStyle(color: Colors.white, fontSize: 16),
      validator: (value) {
        if (value == null || value.isEmpty)
          return "Please enter a password";
        if (value.length < 4 || value.length > 8)
          return "Password must be 4-8 characters long";
        return null;
      },
    ),
    const SizedBox(height: 24),

    // Role Selection
    DropdownButtonFormField<String>(
      value: _selectedRole,
      decoration: _buildInputDecoration(
        "Select Role", Icons.person_outline),
```

```
dropdownColor: Colors.deepPurple,
style: const TextStyle(color: Colors.white),
items: const [
  DropdownMenuItem(
    value: "student",
    child: Text("Student",
      style: TextStyle(
        color: Colors.white, fontSize: 16))),
  DropdownMenuItem(
    value: "admin",
    child: Text("Admin",
      style: TextStyle(
        color: Colors.white, fontSize: 16))),
],
onChanged: (value) {
  setState() {
    _selectedRole = value!;
  });
},
),

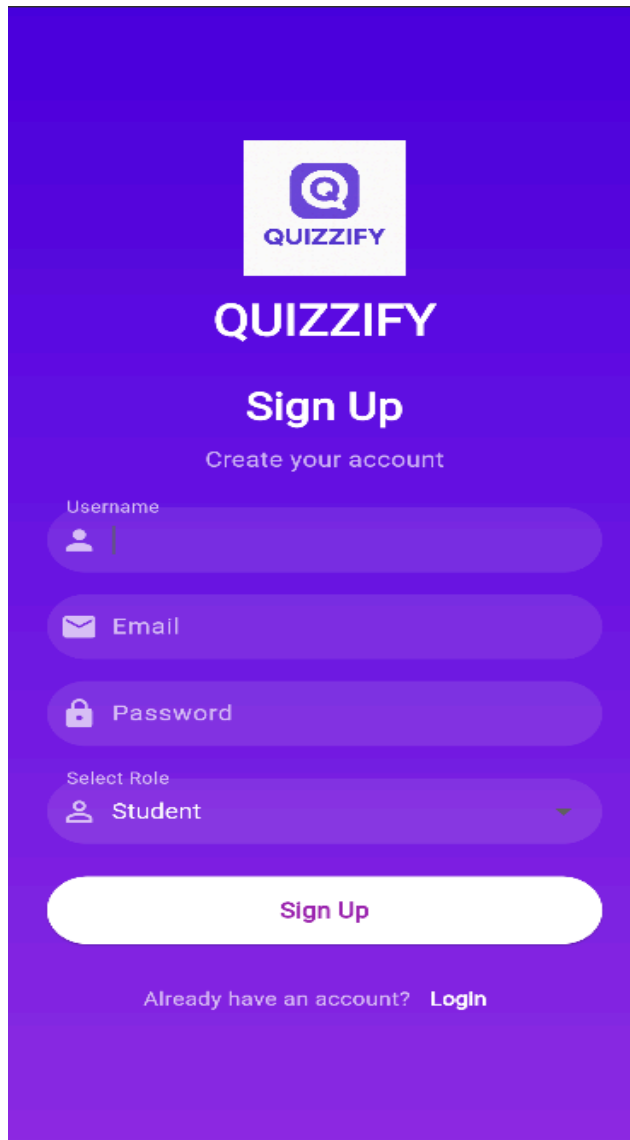
const SizedBox(height: 24),

// Sign Up Button
SizedBox(
  width: double.infinity,
  height: 50,
  child: ElevatedButton(
    onPressed: _isLoading ? null : _signUp,
    style: ElevatedButton.styleFrom(
      backgroundColor: Colors.white,
      shape: RoundedRectangleBorder(
        borderRadius: BorderRadius.circular(30)),
    ),
  child: _isLoading
    ? const CircularProgressIndicator()
    : const Text(
      "Sign Up",
      style: TextStyle(
        color: Colors.purple,
        fontWeight: FontWeight.bold,
```



```
InputDecoration _buildInputDecoration(String label, IconData icon) {
  return InputDecoration(
    labelText: label,
    labelStyle: const TextStyle(color: Colors.white70),
    filled: true,
```

```
fillColor: Colors.white.withOpacity(0.1),  
prefixIcon: Icon(icon, color: Colors.white70),  
border: OutlineInputBorder(  
  borderRadius: BorderRadius.circular(30), borderSide: BorderSide.none),  
);  
}  
}
```

The image shows a mobile application screen for signing up. At the top, there is a white square logo with a blue 'Q' and the word 'QUIZZIFY' in blue. Below the logo, the word 'QUIZZIFY' is written in large, bold, white capital letters. Underneath, 'Sign Up' is written in bold white text, followed by 'Create your account' in a smaller white font. The form consists of four input fields: 'Username' with a person icon, 'Email' with an envelope icon, 'Password' with a lock icon, and 'Select Role' with a dropdown arrow. The 'Select Role' dropdown is currently showing 'Student' with a person icon. Below the form is a large white button with the text 'Sign Up' in blue. At the bottom, there is a link that says 'Already have an account? Login' in white text.**Conclusion:**

The SignUpPage in your Quiz App demonstrates an interactive form using Flutter's Form and TextFormField widgets. It efficiently handles user input through controllers and validations for fields like username, email, and password. Dropdowns add interactivity by allowing role selection within the form. The form ensures user input is validated before Firebase sign-up is triggered. Overall, it provides a user-friendly and functional interface for account creation.

Experiment 05: To apply navigation, routing and gestures in Flutter App**Theory:**

In Flutter, navigation, routing, and gestures are core elements that contribute to an app's interactivity and flow.

The provided HomePage code effectively demonstrates all three concepts:

1. **What is Navigation in Flutter**
Navigation in Flutter allows switching between different screens (pages) within the app. It manages the user flow and screen hierarchy.
2. **Routing Definition**
Routing refers to the process of defining and managing the paths (routes) that lead to different screens in the app. Each route corresponds to a screen or widget.
3. **Navigator Class**
Flutter provides the Navigator class to manage a stack-based history of routes. It allows pushing new routes onto the stack and popping them when going back.
4. **Named Routes vs Anonymous Routes**
 - Anonymous Routes: Defined inline when pushing a new route.
 - Named Routes: Predefined in the app and accessed using a unique route name.
5. **Route Settings**
RouteSettings is used to pass information to a new route such as name and arguments.
6. **MaterialPageRoute**
A common way to define routes in Flutter, used for material design transitions between screens.
7. **Defining Routes in main.dart**
Named routes are defined in the MaterialApp widget using the routes parameter, which maps route names to widget builders.
8. **Initial Route**
The first screen to be displayed is set using the initialRoute property of MaterialApp.
9. **Push and Pop Operations**

- Navigator.push() adds a new route to the stack.
- Navigator.pop() removes the current route and returns to the previous one.

10. Passing Data Between Screens

Data can be passed to another screen via constructors or route arguments and retrieved when the new screen loads.

What are Gestures

Gestures are user interactions like taps, swipes, long presses, and drags that can be detected and responded to in Flutter apps.

GestureDetector Widget

GestureDetector is a core widget in Flutter used to detect and respond to gestures. It wraps any widget and provides callback functions for various gesture events.

Common Gesture Events

- onTap: Detects tap on the widget.
- onDoubleTap: Detects double-tap.
- onLongPress: Detects long press.
- onPanStart, onPanUpdate, onPanEnd: Detect drag gestures.

InkWell and InkResponse Widgets

These widgets provide ripple effect and gesture detection in material design apps. They are often used with buttons and tappable cards.

Custom Gesture Handling

Flutter allows combining gesture callbacks with logic to create custom interactions such as dragging objects or resizing UI elements.

Gesture Arena in Flutter

Flutter uses a gesture recognition system called the gesture arena to resolve conflicts when multiple gestures compete for the same event.

Pointer Events

Flutter also supports low-level gesture handling using pointer events like `PointerDownEvent`, `PointerMoveEvent`, and `PointerUpEvent`.

Interactive Widgets

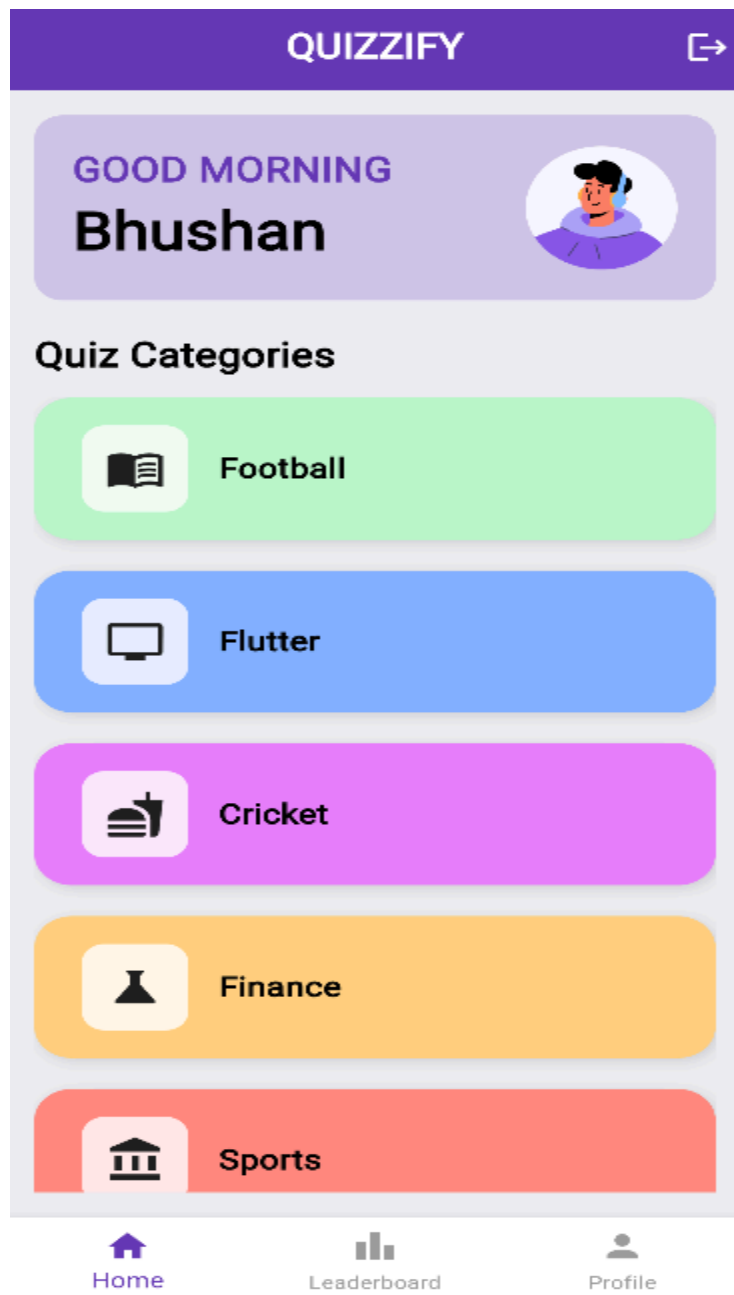
Many Flutter widgets come with built-in gesture support, such as `ElevatedButton`, `TextButton`, `ListTile`, and `Dismissible`.

Scroll and Swipe Gestures

Flutter provides specialized widgets for handling scrolling (`ListView`, `SingleChildScrollView`) and swiping (`Dismissible`, `PageView`).

Importance of Gestures

Gestures improve user experience by enabling intuitive interaction patterns, supporting accessibility, and allowing natural app navigation.

**Conclusion :**

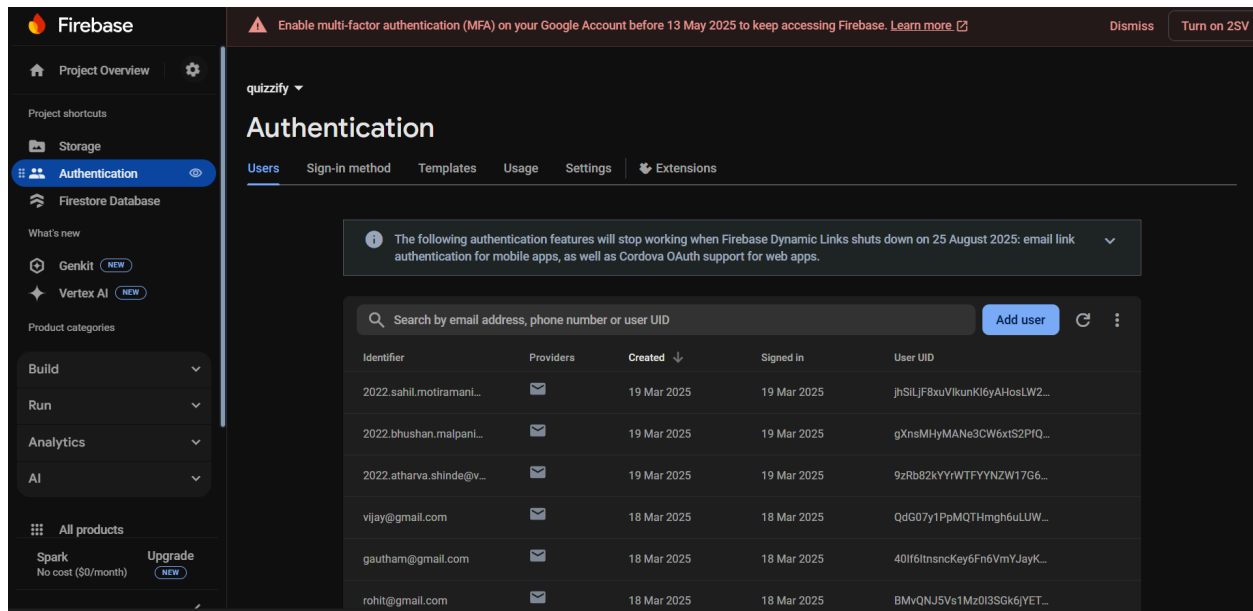
This Flutter app successfully demonstrates routing and navigation using Navigator and MaterialPageRoute. Gesture detection is implemented using GestureDetector to enable navigation upon tapping quiz categories. The BottomNavigationBar enhances seamless switching between Home, Leaderboard, and Profile pages. User data and dynamic content from Firestore are integrated with stateful widgets and real-time updates. Overall, this experiment shows effective use of navigation, routing, and gesture handling for interactive app development.

Experiment 06: To connect flutter with firebase**Theory:**

Firebase is a powerful Backend-as-a-Service (BaaS) platform provided by Google that supports features like authentication, real-time database, Firestore, cloud functions, and more. Connecting Firebase with a Flutter app allows developers to integrate these backend services easily. Flutter interacts with Firebase using plugins like `firebase_core`, `firebase_auth`, and `cloud_firestore`. This integration enables building scalable, real-time, and secure mobile applications.

Steps to Connect Flutter with Firebase:

1. **Create a Firebase Project:**
 - Go to Firebase Console → Create Project → Register App (Android/iOS).
2. **Register your App:**
 - For Android: Add your app's package name (`com.example.appname`), download `google-services.json` and place it in `android/app`.
3. **Add Firebase SDK:**
 - In `android/build.gradle`, add Google services classpath.
 - In `android/app/build.gradle`, apply the `google-services` plugin.
4. **Add Dependencies** in `pubspec.yaml`:
5. **Initialize Firebase in Flutter:**
 - In `main.dart`, ensure `WidgetsFlutterBinding.ensureInitialized()` and await `Firebase.initializeApp();`.
6. **Set up Firebase Authentication and Firestore:**
 - Use `FirebaseAuth` for login/register and `FirebaseFirestore` to read/write data.
7. **Run & Test:**
 - Use a real or virtual device to run the app and test Firebase functions like sign-in, data fetch, etc.



Conclusion:

Flutter and Firebase integration enables real-time backend features like user authentication and cloud storage. Using plugins like `firebase_core`, `firebase_auth`, and `cloud_firestore`, Flutter apps can securely access Firebase services. The connection setup involves configuration in both the Firebase console and the Flutter project. This experiment allows seamless two-way communication between the app UI and backend. Overall, Firebase provides a scalable and efficient backend for Flutter applications.

Experiment - 7

AIM:

To write metadata of your PWA in a Web App Manifest file to enable the “Add to Homescreen” feature.

Theory:

1. What is a Web App Manifest?

A Web App Manifest is a small JSON file that gives important info about your web app — like its name, icons, starting page, colors, and how it should look when opened from a phone’s home screen. It helps the browser know how your app should behave when users install it.

2. How to Enable "Add to Homescreen"

To let users add your app to their home screen, make sure:

- You link the manifest file inside the <head> tag of your HTML.
- Your website is served over HTTPS (secure connection).
- You have a service worker set up to handle offline support and caching.

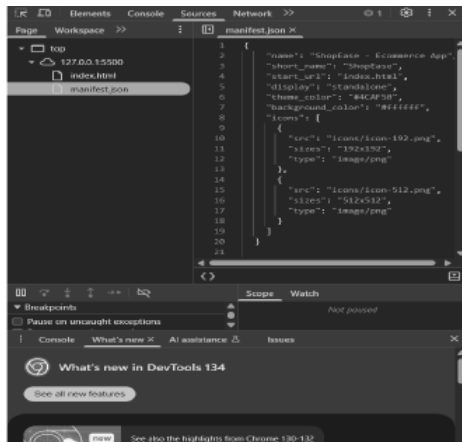
3. Why Use PWAs? (Benefits)

- Gives a feel just like a real mobile app.
- Helps users stay engaged and come back more often.
- Works offline or in weak internet areas.
- Lets you show your own app icon and splash screen, boosting your brand.

```
{  
"name": "Norway Chess Website Clone",  
"short_name": "ShopEase",
```

Experiment - 7

```
"start_url": "/index.html",
"display": "standalone",
"background_color": "#ffffff",
"theme_color": "#4CAF50",
"icons": [
{
"src": "icons/icon-192x192.png",
"sizes": "192x192",
"type": "image/png"
},
{
"src": "icons/icon-512x512.png",
"sizes": "512x512",
"type": "image/png"
}
```



Conclusion:

Adding a Web App Manifest file with proper metadata is essential for enabling the “Add to Home Screen” feature in PWAs. It improves usability, promotes user retention, and delivers a seamless mobile-like experience.

Experiment - 8

AIM:

To code and register a service worker, and complete the install and activation process for a new service worker for the PWA.

Theory:

1. What is a Service Worker?

A **Service Worker** is a type of background script that runs in the browser, but **separately from your web pages**. It helps your web app work even when there's **no internet**, and enables features like **background sync** and **push notifications**. It's a key part of making **Progressive Web Apps (PWAs)** more reliable.

It works like a **middleman between the app and the internet**, handling how requests are made and what to do when the app is offline — such as showing cached content.

2. Service Worker Lifecycle

The service worker goes through **three main steps**:

- **Registration:** The service worker file is linked to the browser from your main code.
- **Installation:** Important files like HTML, CSS, and images are saved (cached) so they can be used later.
- **Activation:** The service worker starts working, takes control of pages, and removes any outdated cache.

3. How Registration Works

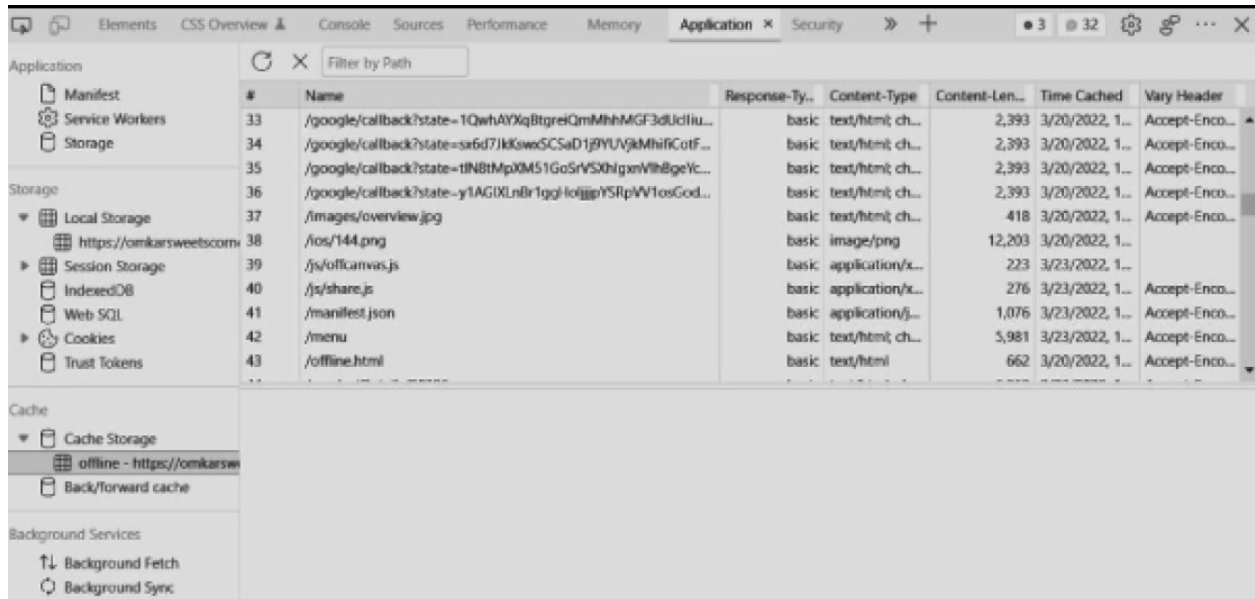
You **register the service worker** in your main JavaScript file. After that, the browser takes care of the rest — like installing and activating it — if everything is done correctly.

4. Why Service Workers Matter for PWAs

- Your app keeps working even when the internet is slow or unavailable.
- It loads faster by using saved (cached) content.

Experiment - 8

- It gives a smoother, more reliable experience, keeping users happy and more likely to return.
- Enables "Add to Homescreen" and background features.



The screenshot shows the Chrome DevTools Application tab. The left sidebar lists categories: Application, Storage, Cache, and Background Services. The main panel displays a table of application data. The table has columns: #, Name, Response-Ty..., Content-Type, Content-Len..., Time Cached, and Vary Header. The data includes various Google callbacks, local storage items, session storage, indexedDB, web SQL, cookies, and trust tokens. The Cache section shows an offline cache for https://omkarsweets.com.

#	Name	Response-Ty...	Content-Type	Content-Len...	Time Cached	Vary Header
33	/google/callback?state=1QwhAYXqBtgreiQmMhMhGF3dUcliu...	basic	text/html; ch...	2,393	3/20/2022, 1...	Accept-Enco...
34	/google/callback?state=sx6d7JkSweSCSaD1j9YUUVjkMhRiCotF...	basic	text/html; ch...	2,393	3/20/2022, 1...	Accept-Enco...
35	/google/callback?state=tIN8tMpXMS1GoSvSXhigxVihBgeYc...	basic	text/html; ch...	2,393	3/20/2022, 1...	Accept-Enco...
36	/google/callback?state=y1AGDXLnBr1ggfIoIppYSRpVW1osGod...	basic	text/html; ch...	2,393	3/20/2022, 1...	Accept-Enco...
37	/images/overview.jpg	basic	text/html; ch...	418	3/20/2022, 1...	Accept-Enco...
38	/ios/144.png	basic	image/png	12,203	3/20/2022, 1...	
39	/js/officamvas.js	basic	application/x...	223	3/23/2022, 1...	
40	/js/share.js	basic	application/x...	276	3/23/2022, 1...	Accept-Enco...
41	/manifest.json	basic	application/j...	1,076	3/23/2022, 1...	Accept-Enco...
42	/menu	basic	text/html; ch...	5,981	3/23/2022, 1...	Accept-Enco...
43	/offline.html	basic	text/html	662	3/20/2022, 1...	Accept-Enco...

Conclusion:

Coding and registering a service worker is crucial in transforming a traditional web app into a fully functional PWA. It enhances user experience by providing offline capabilities, faster performance, and better engagement for applications.

Experiment - 9

AIM:

To implement Service Worker events like fetch, sync, and push for PWA.

Theory:

How Service Workers Improve PWAs

Service Workers boost the power of Progressive Web Apps (PWAs) by adding smart features like offline support, background syncing, and push notifications. These features work through special service worker events: fetch, sync, and push.

1. Fetch Event

This event happens whenever the app asks for something from the internet (like a webpage or image).

The service worker can catch these requests and either serve saved (cached) files or fetch new ones online.

This helps the app load faster and even work offline when there's no internet.

2. Sync Event

This event kicks in when the device gets back online after being offline.

It lets the service worker try again to send things that failed earlier (like a form you submitted or items added to a cart).

It helps keep data correct and up-to-date, even with bad internet.

3. Push Event

This event happens when the server sends a message to the service worker.

It allows the app to show notifications, like new offers or order updates, even if the app isn't open in the browser.

It helps keep users informed in real time.

4. Why This Matters for PWAs

- **Fetch** keeps key pages and files ready to use, even offline.
- **Sync** ensures that your order or cart stays accurate even after lost connections.
- **Push** keeps users updated with **real-time alerts**, like special deals or important info.

Experiment - 9

http://localhost:3000/ - deleted Network requests Update Unregister

Source

Status

Clients

Push

Conclusion:

Implementing fetch, sync, and push events in a service worker greatly improves the functionality and reliability of an PWA. It ensures better performance, user engagement, and seamless experience across network conditions.

Experiment - 10

AIM:

To study and implement deployment of PWA to GitHub Pages.

Theory:

1. What is GitHub Pages?

GitHub Pages is a free hosting platform that lets developers publish websites straight from a GitHub repository. It's perfect for putting up static websites, like Progressive Web Apps (PWAs), with an easy and quick setup.

2. How to Deploy a PWA on GitHub Pages

To publish a PWA using GitHub Pages, follow these steps:

- Upload your project to a GitHub repository.
- Use a command like `npm run build` to create static files (HTML, CSS, JS) for deployment.
- Make sure these build files are served from the correct branch, usually `gh-pages`.

3. Main Steps in Deployment

- Set Up the Repository: Push your PWA code to GitHub.
- Build the Project: Create production-ready static files.
- Configure Deployment: Use tools like the `gh-pages` npm package or GitHub Actions to automate the process.
- Set the Homepage URL: In your `package.json`, add the correct homepage link so your app loads resources properly.

4. Why Use GitHub Pages?













- It's free and beginner-friendly.
- No need for complicated server settings.

Experiment - 10

- Uses GitHub's fast CDN for quick loading everywhere.
- Great for hosting PWAs or showcasing front-end projects.

5. Why It Matters for PWAs

Hosting your PWA on GitHub Pages lets people access and install your app from any browser. It gives them a chance to try out features, browse content, and experience the app just like a real-world product.

 android	Fresh start	2 weeks ago
 assets/flags	Fresh start	2 weeks ago
 ios	Fresh start	2 weeks ago
 lib	more progress	2 weeks ago
 linux	Fresh start	2 weeks ago
 macos	Fresh start	2 weeks ago
 test	Fresh start	2 weeks ago
 web	Fresh start	2 weeks ago
 windows	Fresh start	2 weeks ago
 .env	Fresh start	2 weeks ago
 .gitignore	Fresh start	2 weeks ago
 .metadata	Fresh start	2 weeks ago

Conclusion:

Deploying an PWA to GitHub Pages provides a quick and reliable way to make the app available online. It simplifies distribution, testing, and user access without the need for traditional web hosting.

Experiment - 11

AIM:

To use Google Lighthouse PWA Analysis Tool to test the PWA functioning.

Theory:

Google Lighthouse is an open-source, automated tool used to audit web applications. It helps developers evaluate and improve the quality of Progressive Web Apps (PWAs) by generating detailed reports based on performance, accessibility, best practices, SEO, and PWA standards.

1. What is the Lighthouse PWA Audit?

Lighthouse checks whether a web app qualifies as a PWA by testing features like offline capability, HTTPS usage, responsive design, and “Add to Homescreen” readiness. The audit provides a score and suggestions for improvement.

2. How to Use Lighthouse

Lighthouse can be accessed through:

- **Chrome DevTools (Audits tab)**
- **Command line (lighthouse <url>)**
- **Lighthouse CI or GitHub Actions**

Users can run audits directly on their live or local PWA by generating a report within seconds.

3. Key Checks for PWA Audit

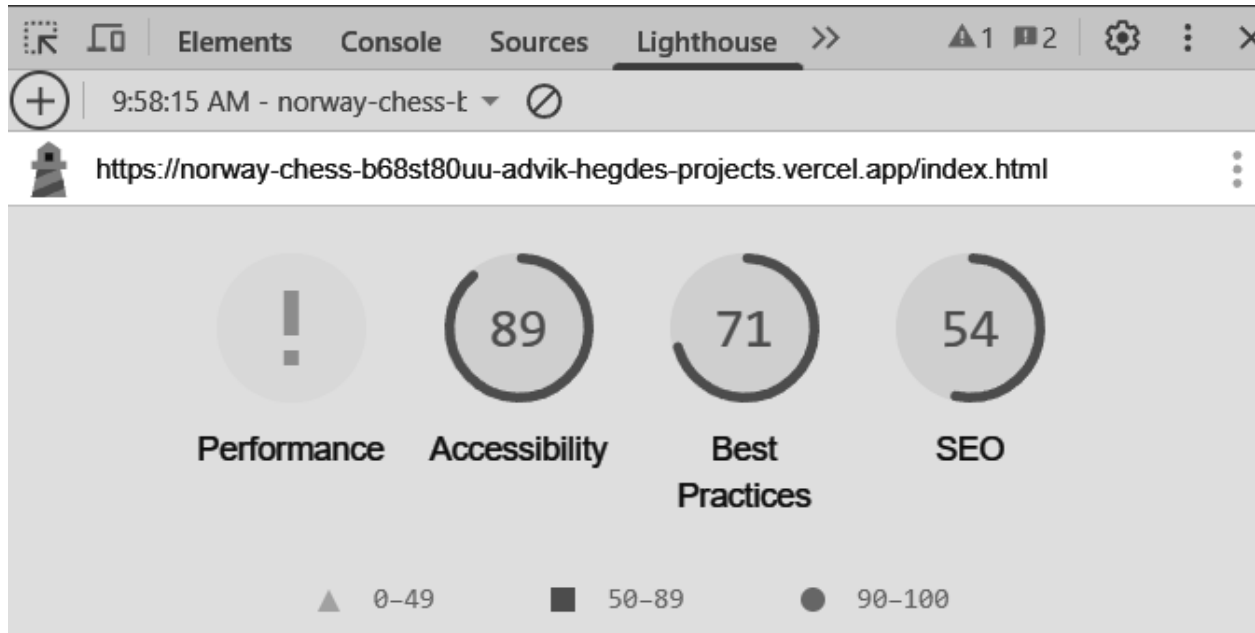
- App is served over HTTPS
- Service worker is registered and active
- Web App Manifest is valid
- Responsive and works offline
- Provides custom splash screen and install prompt

4. Benefits for E-commerce PWAs

- Ensures the app delivers a fast, reliable, and engaging experience

Experiment - 11

- Identifies and fixes performance and accessibility issues
- Helps in meeting PWA requirements for better user retention



Conclusion:

Using Google Lighthouse for PWA analysis ensures the E-commerce app meets modern web standards. It improves reliability, usability, and performance, leading to a better user experience and increased engagement.