

Application Development using Flutter And Dart

[Document subtitle]



[Date]

[Company name]

[Company address]

**Introduction:**

**Flutter** is a front-end framework focused on building the user interface and client-side logic of applications. It is often used in conjunction with other back-end technologies to create complete applications.

**Cross-Platform Development:** Flutter allows developers to create apps that run on multiple platforms, like Android, iOS, web, and desktop, using a single codebase. This means you write the code once, and it works on many devices.

**Widgets and UI:** Flutter uses a system of customizable widgets to create user interfaces. These widgets are like building blocks that you can combine to design the look and feel of your app.

**Dart Programming Language:** Flutter apps are written in Dart, a programming language also created by Google. It’s designed to be easy to learn and efficient for building apps.

**Hot Reload:** One of Flutter’s standout features is "hot reload," which allows developers to see changes in the code instantly in the app. This speed up the development process by letting developers quickly test and see their changes.

**Some Application Which we can make on Flutter**

1. **Mobile Apps**
2. **Web Applications**
3. Desktop Applications
4. Embedded Devices
5. Game Development
6. Enterprise Applications
7. Multimedia Applications
8. Finance Applications

Flutter excels in front-end development, it does not handle back-end tasks like database management, server-side logic, or API development. For full-stack development, Flutter is often paired with back-end technologies such as:

**Firebase:** A popular choice for back-end as a service (BaaS) with features like authentication, cloud storage, and real-time databases.

**Custom APIs:** Built with Node.js, Django, Laravel, or other server-side frameworks.

HOW FLUTTER AND DART WORK TOGETHER IN AN APP

**Example: Counter App**

**App Overview:**

This app has a button that, when pressed, increases a counter. The counter value is displayed on the screen.

**Roles of Flutter and Dart**

**1. Flutter (UI Framework)**

**Purpose**: Flutter is responsible for the user interface (UI) and visual elements of the app.

UI Components in Flutter:

**MaterialApp:** Sets up the basic app structure with Material Design.

**Scaffold:** Provides the app’s basic layout structure, including the AppBar and body.

**Text:** Displays the counter value on the screen.

**ElevatedButton:** Represents the button that the user presses to increase the counter.

**2. Dart (Programming Language)**

**Purpose:** Dart is used for the logic and state management behind the app’s UI.

**Logic in Dart:**

**State Management:** Dart handles the counter’s value and updates the UI when the button is pressed.

**Event Handling:** Dart defines what happens when the button is pressed (e.g., incrementing the counter).

*Flutter is used for creating the UI and layout of the app, while Dart handles the logic and state management. They work together within a single framework, simplifying development by using one language for both UI and logic.*

**Installation Guide**

**1. System Requirements**

Before installing Flutter, ensure that your system meets the following requirements:

**Operating System:** Windows 10 or later (64-bit), macOS, or Linux.

**Disk Space:** At least 2.5 GB of free disk space (excluding IDE/tools).

**Tools:** Git for version control.

**2. Download Flutter SDK**

**Download Flutter SDK:**

Go to the Flutter official website and download the latest stable version for Windows.

**Extract the Flutter SDK:**

Extract the downloaded .zip file to a location of your choice (e.g., C:\flutter or D:\flutter).

**Update Path Variable:**

* Search for "Environment Variables" in the Start menu.
* Click on Environment Variables in the System Properties window.
* Under System variables, select Path and click Edit.
* Click New and add the path to the flutter\bin directory (e.g., C:\flutter\bin or D:\flutter\bin).
* Click OK to apply the changes.

**Run Flutter Doctor**

After installation, run the flutter doctor command to ensure everything is set up correctly.

**3. Setting up an Editor**

You can use any text editor or IDE for Flutter development, but the most common ones are:

**Visual Studio Code:** Install the *Flutter and Dart* extensions.

**Android Studio:** Flutter plugin is available via the plugin marketplace.

Now just press f5 to verify that it is working on chrome or browser or not.

-----------------------------------------xxxxxxxxxxxxxxxxxxxxxxx-----------------------------------------------------

**Flutter project structure**

**1. Root Directory**

**android/:** Contains the Android-specific code and configuration. This is where you manage native Android code and settings, such as permissions, Gradle scripts, and the AndroidManifest.xml file.

**ios/:** Similar to the Android directory, this contains the iOS-specific code and configuration. You manage your Xcode project, Info.plist, and iOS-specific settings here.

**lib/:** This is where most of your Dart code resides. The main.dart file, which serves as the entry point of the application, is located here. You can create additional Dart files and directories here to organize your app’s code (e.g., screens/, models/, widgets/, etc.).

**test/:** Contains unit and widget tests. Flutter uses the Dart testing framework, and this directory is where you write tests to ensure the correctness of your application.

**web/ (optional):** If you are building a Flutter web application, this directory contains web-specific code, including the HTML, JavaScript, and other assets needed to run the application in a web browser.

**macos/ (optional):** For macOS-specific code and configuration, this directory is used when you build macOS desktop applications.

**linux/ (optional):** Contains Linux-specific code and configuration files for building Linux desktop applications.

**windows/ (optional):** For Windows-specific code and configurations when building Windows desktop applications.

**build/:** This directory is automatically generated by Flutter and contains the compiled output of your project. You typically don't interact with this directory directly.

**pubspec.yaml:** A critical configuration file that manages the project’s dependencies, assets, and other settings. It is used to define the dependencies for the project (both from pub.dev and local packages), and you can also declare assets like images and fonts.

**pubspec.lock:** Generated after running flutter pub get, this file locks the versions of the dependencies to ensure consistency across environments.

**.dart\_tool/:** A hidden directory that stores tool-specific files, including the package resolution cache and other build-related files. You typically don't need to interact with this directory.

**.gitignore:** Specifies files and directories that Git should ignore when committing to the repository. This usually includes build files, system-specific files, and other generated files.

**.metadata:** Stores metadata about the project, including the Flutter SDK version used to create the project.

**README.md:** A markdown file where you can provide an overview of your project, instructions on how to set up and run it, and other relevant information.

**analysis\_options.yaml (optional):** Used to customize static analysis rules for your Dart code. You can specify linter rules and other analysis options here.

**2. Common Additional Directories**

**assets/:** This is where you store images, fonts, JSON files, and other non-code assets that your app needs.

**fonts/:** If you’re using custom fonts, they are usually stored here and referenced in the pubspec.yaml file.

**3. Important Files**

**main.dart:** Located in the lib/ directory, this file serves as the entry point for the application. It typically contains the void main() function and the root widget of your app.

**flutter\_launcher\_icons.yaml (optional):** If you are using the flutter\_launcher\_icons package, this file is used to configure app icons for different platforms.

**4. Platform-Specific Considerations**

**Android (android/):** Contains files like build.gradle, AndroidManifest.xml, and res/ for managing Android-specific configurations and resources.

**iOS (ios/):** Includes the Info.plist file, Xcode project settings, and other iOS-specific files.

**Web (web/):** Contains index.html, JavaScript files, and other resources necessary for running a Flutter app in a web environment.

First App: Basic Understanding of Flutter and Dart

1.Open your created project on vscode.

2.find main.dart as all work will be loaded from this file as we did in c#.

3. first thing is to import material.dart file.

**Material.dart**

"If I want to use Flutter's built-in tools to make my app look like a modern Android app with a nice design." It allows you to use widgets like buttons, text fields, and other UI components easily. By importing material. dart, you get access to a lot of useful widgets and tools to build a user interface, like buttons, text, app bars, etc., that follow Google's Material Design guidelines.

4.after importing now create a main function where all will be called :

void main() {

}

Next step is making a class where we will create code of our first app.but first get the concept of class in flutter.

**Widget:**

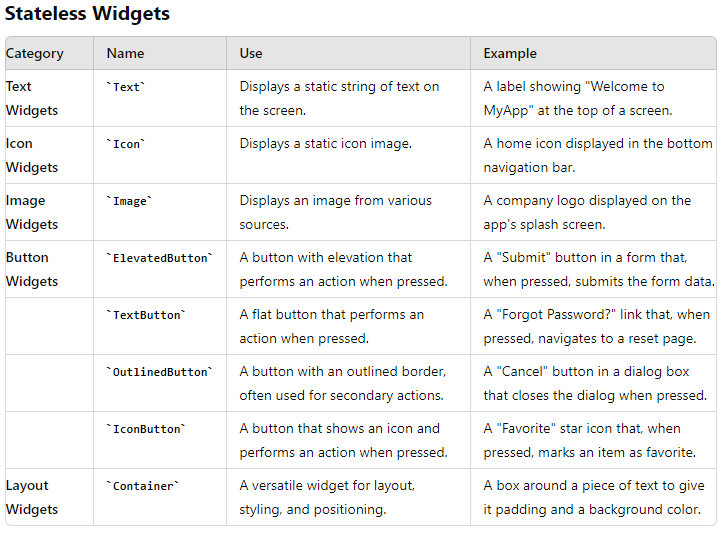
A widget is a **Dart class** that represents a part of the user interface. In Flutter, widgets are the basic building blocks of the app's UI. Everything you see on the screen — buttons, text, images, layouts — is a widget. In Flutter, you create custom widgets by writing Dart classes.

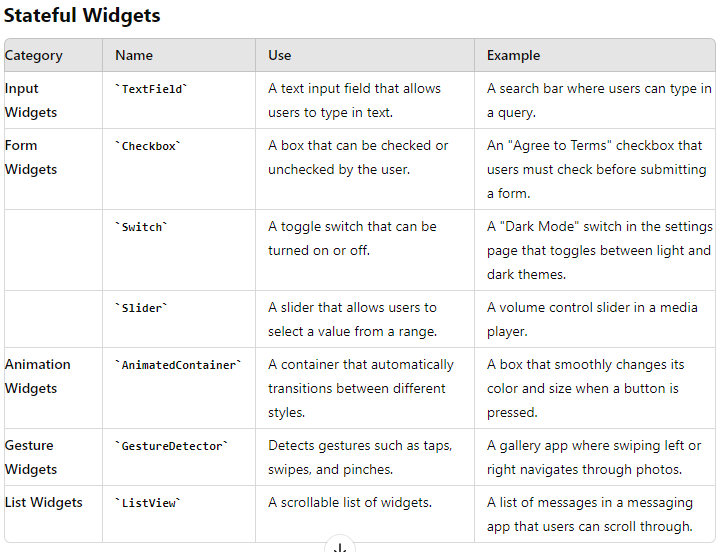
**Widget** **categories:**

In Flutter, widgets are primarily divided into two categories based on whether they can change (**stateful**) or remain constant (**stateless**). Here’s how the built-in widgets can be categorized:

**Stateless Widgets** are used when the UI elements do not need to change or respond to user input dynamically.

**Stateful Widgets** are used when the UI needs to change based on user interaction, data changes, or animations.





**Custom Widget in Flutter:**

**Define a Class:**

Your custom widget is a Dart class that extends either StatelessWidget or StatefulWidget, depending on whether your widget needs to manage state.

A custom widget in Flutter is a widget that you create yourself, usually by combining other widgets or by defining new behavior or styling that isn't available in the built-in widgets.

Creating custom widgets allows you to reuse code, maintain consistency across your app, and encapsulate specific functionality or design elements.

**Task: Display a textbox on Screen**

You should extend **StatelessWidget** or **StatefulWidget** only when you're creating a class meant to be used as part of the UI.

when you're creating a UI component (a widget) that will be displayed on the screen, it should extend either **StatelessWidget or StatefulWidget** to be recognized as a widget by the Flutter framework.

**MyApp class in a Flutter**

the MyApp class typically serves as the **entry point and configuration hub** for the entire application. This class serves as the central point where your app’s setup begins, making it a crucial part of your Flutter app's structure.

what kind of information we usually put in MyApp?

**Theme:** Defines the look and feel of the app globally, including colors, fonts, and styles for widgets.

**Title:** The title of the app, which might be used by the system in places like the app switcher.

**Localization:** Configures the languages and locale settings if your app supports multiple languages.

**Initial Route:** Specifies the first screen the app should display when launched.

**Home Widget:** The main or initial screen that the app shows when launched. This is usually the most important or frequently used screen in the app.

**Routes:** Defines the available screens (routes) in your app and how to navigate between them. This can be managed using named routes or other navigation strategies.

MaterialApp

MaterialApp is like a toolkit that provides a lot of useful features and saves you from having to manually set up each of these aspects yourself.

Using MaterialApp in Flutter is like using a standard template or toolkit that sets up many of the common features needed in most mobile apps. It provides several built-in conveniences that help streamline app development.

*MaterialApp applies Material Design principles, which include consistent colors, fonts, and user interface elements.*

**Structure:** It wraps your entire app and defines how it looks and behaves.

**Theme:** Sets the overall color, font, and style for your app.

**Home:** Specifies the first screen your app displays when it launches.

**Routes:** Manages navigation, allowing you to move between different screens.

   return MaterialApp(

      title: 'Simple TextBox Demo', //The name of your app. It might be shown in system UI or recent apps lists.

      theme: ThemeData(

        primarySwatch: Colors.blue, //Defines the color scheme and overall look of your app. In the example, it sets the primary color to blue.

      ),

      home: SimpleTextBoxPage(), //Specifies the first screen that users will see when they open your app. In the example, SimpleTextBoxPage is the first screen.

    );

**Build Method**

**Widget (Return Type):** This indicates that the build method returns a widget. In this case, it returns a MaterialApp widget.

**build (Method):** This method constructs and returns the widget tree for this part of the UI.

**BuildContext (Parameter):** This provides information about where this widget is located in the widget tree and gives access to various inherited properties like theme data.

*The build method can return any widget that defines what should be displayed on the screen.*

Common widgets like **MaterialApp** and **Scaffold** are used for structuring the overall app and individual screens, respectively.

Other widgets like **Container**, **Column**, **Row**, etc., are used to create specific layouts and designs within those structures.

**Now Simple MyApp Class Altogether**

class MyApp extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return MaterialApp(

      title: 'Simple TextBox Demo',

      theme: ThemeData(

        primarySwatch: Colors.brown,

      ),

      home: SimpleTextBoxPage(),

    );

  }

}

class MyApp extends StatelessWidget {

In Flutter, everything is a widget. By creating a class that extends StatelessWidget, you're defining a widget that does not change its state over time. This MyApp widget will be the starting point of your app.

@override

It tells the Dart compiler that you're intentionally overriding a method that exists in the parent class (StatelessWidget in this case).

Widget build(BuildContext context) {

The build method is required for any widget. It’s the method that tells Flutter what to display on the screen. BuildContext provides information about where this widget fits in the widget tree and allows access to things like the theme and other inherited properties.

return MaterialApp(

You're telling Flutter that the MyApp widget will create a MaterialApp. A MaterialApp is a special widget that wraps the whole app and provides many built-in functionalities like navigation, theming, etc. It's like the foundation for building a Material Design app.

title: 'Simple TextBox Demo',

This is a named parameter in MaterialApp.The title is the name of your app. It might be displayed by the operating system in some places, like the task switcher.

theme: ThemeData(

*ThemeData in Flutter is a class that contains the configuration for the overall visual theme of a Material app. It defines the colors, fonts, shapes, and other styling aspects that will be applied across the entire app. Think of it as a blueprint that dictates how all the UI elements in your app should look and feel.*

specifying the theme of your app using ThemeData. A theme defines the look and feel of your app, including colors, fonts, and other visual aspects.

primarySwatch: Colors.brown,

primarySwatch sets the primary color scheme of your app to brown. This will influence the color of many widgets across your app, like the AppBar, buttons, etc.

home: SimpleTextBoxPage(),

Another named parameter of MaterialApp.home sets the default or main screen of your app. When your app starts, it will display whatever widget you assign to home. In this case, SimpleTextBoxPage() is another widget (defined elsewhere) that will be shown as the main screen.

**Widget Tree**

In Flutter, a widget tree is a hierarchical structure that represents how widgets (UI components) are organized and nested within each other to create the user interface (UI) of an app. Think of it like a family tree, where each widget can have one or more child widgets, and those children can have their own children, and so on.

****

**Properties we can use of Each Widgets:**

**MaterialApp Properties**

* title: The title of the app, displayed in the app bar.
* home: The initial widget displayed when the app starts.
* routes: A map defining named routes for navigation.
* theme: The overall theme for the app, including colors, fonts, and styles.
* darkTheme: The theme for dark mode.

**MaterialApp Navigation Properties:**

* initialRoute: The initial route to navigate to.
* onGenerateRoute: A function that generates routes based on the route name.
* onUnknownRoute: A function that handles unknown routes.

**Code Example:**

class MyApp extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return MaterialApp(

      title: 'My Flutter App',

theme: ThemeData(

        primarySwatch: Colors.blue,

        brightness: Brightness.light,

      ),

      darkTheme: ThemeData(

        brightness: Brightness.dark,

        primarySwatch: Colors.deepOrange,

      ),

      home: HomeScreen(),  // Initial widget displayed when the app starts

      routes: {

        '/home': (context) => HomeScreen(),

        '/about': (context) => AboutScreen(),

      },

);

  }

}

**Home Properties**

****

**Scaffold:**

Provides a basic structure for the app, implementing Material Design layout principles.

It contains areas for AppBar, body, floatingActionButton, and more, allowing for easy implementation of common layouts. It is like a blank canvas that provides several default visual elements for your app's UI.

**Container:**

A versatile widget that can hold a single child and apply various properties like padding, margin, color, and decoration.

Useful for creating layouts and controlling the appearance of UI elements.

**Center:**

A widget that centers its child both vertically and horizontally within the available space.

Great for simple layouts where you want to center a widget on the screen.

**ListView:**

A scrollable list of widgets that can be vertically or horizontally arranged.

Ideal for displaying collections of items that may exceed the screen size.

**GridView:**

A scrollable grid of widgets, allowing you to create a layout that displays items in a grid format.

Useful for displaying images, cards, or other content that benefits from a grid layout.

**Stack:**

A widget that overlays its children on top of each other, allowing for complex layouts.

Useful for creating effects like overlapping elements or positioning widgets relative to each other.

**PageView:**

A widget that allows users to swipe between different pages or views.

Commonly used for onboarding screens or image galleries.

**Form:**

A widget that groups multiple form fields and handles form validation.

Useful for input forms where you need to collect user data.

class MyApp extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return MaterialApp(

      title: 'My Flutter App',

      home: Scaffold(

        appBar: AppBar(

          title: Text('AppBar Title'),

        ),

        // Directly included as children of home

        Container(

          // Container properties

        ),

        Center(

          child: Text('Centered Text'),

        ),

        ListView(

          // ListView properties

        ),

        GridView(

          // GridView properties

        ),

        Stack(

          children: [

            // Stack children

          ],

        ),

        PageView(

          children: [

            // PageView children

          ],

        ),

        Form(

          child: Column(

            children: [

              // Form fields

            ],

          ),

        ),

        floatingActionButton: FloatingActionButton(

          onPressed: () {

            // Action for the button

          },

          child: Icon(Icons.add),

        ),

        drawer: Drawer(

          // Drawer contents

        ),

        bottomNavigationBar: BottomNavigationBar(

          items: [

            BottomNavigationBarItem(icon: Icon(Icons.home), label: 'Home'),

            BottomNavigationBarItem(icon: Icon(Icons.settings), label: 'Settings'),

          ],

        ),

      ),

    );

  }

}

**Scaffold properties**

* appBar: Displays a top bar, typically an AppBar widget for title, actions, and navigation.
* body: The primary content of the scaffold, placed below the appBar.
* floatingActionButton: A button that floats above the body, commonly used for primary actions.
* floatingActionButtonLocation: Determines the position of the floatingActionButton on the screen.
* drawer: A sliding panel from the left, usually used for navigation.
* endDrawer: A sliding panel from the right side of the screen.
* bottomNavigationBar: A widget at the bottom for navigating between different views.
* bottomSheet: A widget that appears at the bottom of the screen.
* backgroundColor: Sets the background color of the Scaffold
* SnackBars: Temporary messages that appear at the bottom of the screen for brief periods.

**Now let’s make a class where we place a textbox and show it on main screen**

class SimpleTextBoxPage extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('Simple TextBox'),

      ),

      body: Padding(

        padding: const EdgeInsets.all(16.0),

        child: TextField(

          decoration: InputDecoration(

            border: OutlineInputBorder(),

            labelText: 'Enter some text',

          ),

        ),

      ),

    );

  }

}

**title: Text('Simple TextBox'),**

* Sets the title of the AppBar to the text "Simple TextBox". This text will be displayed in the center of the AppBar.
* This is like putting a nameplate at the top of the page that tells the user what this page is about.

**body: Padding(**

The body parameter of the Scaffold defines the main content of the page. Here, Padding is used to wrap the content with some space around it, defined by the padding parameter.

Imagine putting a picture inside a frame with some space between the picture and the frame itself. This space is the padding.

**padding: const EdgeInsets.all(16.0),**

* Specifies the amount of padding around the child widget. EdgeInsets.all(16.0) means there will be 16 pixels of padding on all sides of the child.
* This is like ensuring there’s a uniform margin around a block of text in a document, so it doesn’t touch the edges.

EdgeInsets in Flutter is a versatile tool to control the spacing around widgets, helping to create visually balanced and well-spaced user interfaces.

**Now move towards Header (AppBar)**

We have two options here define AppBar on each page or create a separate class for it so we can use it on multiple screens. Here I am opting second option (create a separate class).

Before moving towards AppBar we have to go through som pre defined rules .

1. We have to tell what would be a size of AppBar is it default or our defined.
2. To keep the height of AppBar(or any other widget) flutter use an interface named as **PreferredSizeWidget**

*The* ***PreferredSizeWidget*** *is an interface in Flutter that allows a widget to tell the system what size it would like to be. The AppBar is a common example of a widget that needs a specific height (the default height of an AppBar is 56 pixels).*

**Why is PreferredSizeWidget necessary?**

* Imagine you're building a shelf for a room. The shelf needs to fit between two walls, and the height of the shelf is very important for fitting the space properly.
* AppBar is like that shelf.
* PreferredSizeWidget is like telling the constructor what the exact height of the shelf should be so it fits perfectly in the room.
* Without specifying the height, the shelf (or in this case, the AppBar) wouldn’t fit properly in the layout.
* If you don't implement the PreferredSizeWidget, Flutter won't know what size to give to your custom AppBar, which might cause layout issues.

**@override**

**Size get preferredSize => Size.fromHeight(kToolbarHeight);**

Let’s understand this line first

**@override**

*The CustomAppBar class implements PreferredSizeWidget, which requires you to override the preferredSize property. By using @override, you're specifying that you're providing your own implementation for the preferredSize property.*

This interface is like a contract. It says, "If you want to make a custom widget (like a custom AppBar), you must provide me with the preferredSize (dimensions of the widget)." It doesn’t care how you do it, but you must do it.

**This is where Size comes in.**

***The Size class is a tool you use to provide the size (width and height). So, when the interface asks, "What's your preferred size?", you use the Size class to answer with actual dimensions.***

***In this case, you are only concerned with the height, because the AppBar always stretches across the width of the screen. That’s why you use Size.fromHeight().***

**get** is creating a computed property, preferredSize, which returns the size when accessed. This allows you to define dynamic or calculated values rather than hardcoded properties.

**preferredSize**

This is the name of the property that you're overriding from the PreferredSizeWidget interface.preferredSize tells the Flutter framework how big your custom widget (the AppBar) should be. Specifically, it determines the widget's height (since AppBar has a fixed width).

**Size.fromHeight**

This is a constructor of the Size class that creates a Size object based on a given height.Size.fromHeight creates a Size object where the height is specified, and the width will be automatically handled based on the available space (the width isn't needed here because the AppBar stretches across the entire width of the screen).

**kToolbarHeight**

* This is a constant in Flutter that represents the standard height of a toolbar (AppBar), which is 56 logical pixels by default.
* kToolbarHeight is a predefined constant that specifies the height of the toolbar (AppBar). By using it here, you're setting the height of your AppBar to the standard 56 pixels.

***kToolbarHeight is providing default size but what if I want to give my own height?***

If you want a custom height, you can change the value inside Size.fromHeight(). You could even say **Size.fromHeight(80)** to make it taller.

import 'package:flutter/material.dart';

class CustomAppBar extends StatelessWidget implements PreferredSizeWidget {

  final String title;

  final List<Widget>? actions;

  final double customHeight; // Added a custom height variable

  CustomAppBar({required this.title, this.actions, this.customHeight = 70.0}); // Default to 70 pixels

  @override

  Widget build(BuildContext context) {

    return AppBar(

      title: Text(title),

      backgroundColor: Colors.brown,

      centerTitle: true,

      leading: Icon(Icons.menu),

      actions: actions,

    );

  }

  @override

  Size get preferredSize => Size.fromHeight(customHeight); // Using custom height here

}

Here is complete code of our AppBar with custom height. Let’s learn it in details and create its instance in our screen.

**final String title;**

Declares a final variable title that stores the text for the title of the AppBar. This text will be passed into the class when you create an instance of CustomAppBar.titke of each screen is different so will be helpful to set as per screen.

**final List<Widget>? actions;**

Declares a nullable list of Widget objects called actions. This allows you to pass in any additional buttons or widgets (like a search button or profile icon) that will be displayed on the right side of the AppBar.

**final double customHeight;**

Declares a variable customHeight to store the height of the AppBar. By default, it is set to 70.0 pixels if not provided. This gives flexibility to set a custom height for the AppBar. If you want default size then can remove this and use the default height (kToolbarHeight).

**CustomAppBar ({required this. title, this.actions, this.customHeight = 70.0});**

This is the constructor for CustomAppBar. It takes in three parameters: title (required), actions (optional), and customHeight (optional with a default of 70.0). The title is required, but actions and customHeight are optional. You can simplify this if custom height is not needed.

**return AppBar (**

This returns a standard AppBar widget, which will be built using the parameters provided in the constructor (like title and actions).

**title: Text(title),**

This sets the title of the AppBar. The title is displayed as text inside the AppBar. It's passed in as a parameter when creating the CustomAppBar.

**centerTitle: true,**

This centers the title text within the AppBar. By default, the title might align to the left or based on platform-specific rules.

**leading: Icon(Icons.menu),**

*The leading widget in an AppBar is placed at the start of the AppBar (usually the left side). It's commonly used for navigation buttons, like a back button or a hamburger menu icon for a navigation drawer.*

This adds an icon (a hamburger menu in this case) to the left of the AppBar. It acts as the leading widget, often used for a navigation drawer button.

**actions: actions,**

*The actions property holds a list of widgets (usually buttons or icons,* *Search button, Profile button, share button) that appear at the end of the AppBar (typically on the right side). These are often used for quick actions like a search button, settings, or a profile icon.*

This adds additional widgets (passed in as actions) to the right side of the AppBar. These could be buttons or icons like a search button, profile button, etc.

**Createstate Method:**

This method is usually use in stateful widgets. stateful widgets can change their state. when we create a widget, it will set in widgets tree. Each time when state changes tree has to make up again which can slow down UI. we use createstate method and assign it our widget now when our widgets state changes it will change whole tree but the our specific widget. It is same like delegates having address of our widgets and change state immediately.

**Single-Child Widgets:** These widgets can hold only one child widget. They are generally used for modifying or wrapping a single widget, like applying padding, alignment, or decoration.

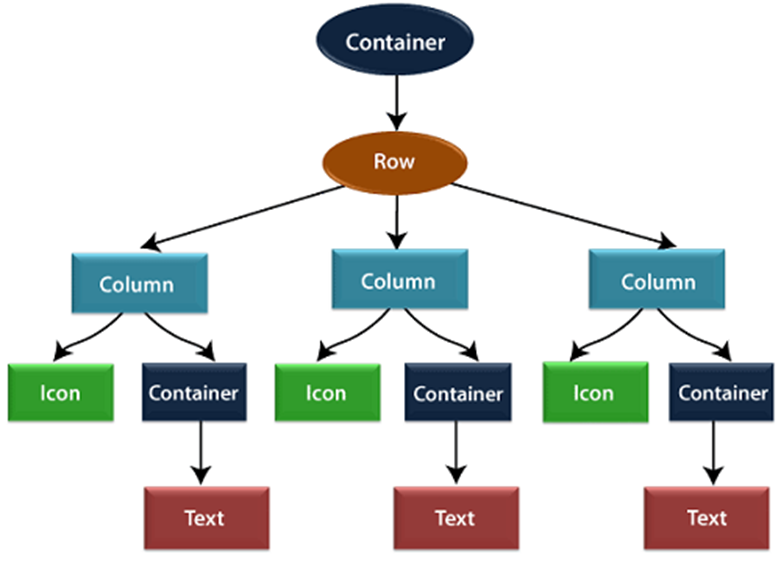
**Multiple-Child Widgets:** These widgets can hold more than one child. They are used for layouts and structures where multiple widgets are required, such as lists, grids, or stacks.



**Container**

In Flutter, a Container is like a box that can hold content. It's one of the most commonly used widgets because it's very flexible. You can think of it as a frame or wrapper around other widgets that allows you to do several things like:

* Add padding, margin, or space around something
* Add a background color or image
* Set the size (width, height) of something
* Align the content inside it



Let’s create a container step by step:

**Create MyApp class for project description:**

**MyApp.dart:**

import 'package:container\_widgets/my\_home\_page.dart';

import 'package:flutter/material.dart';

class MyApp extends StatelessWidget {

  // This widget is the root of your application.

  @override

  Widget build(BuildContext context) {

    return MaterialApp(

      title: 'Flutter Demo',

      home: MyHomePage(),

    );

  }

}

**Create state for our container widget:**

import 'package:flutter/material.dart';

import 'my\_home\_page\_state.dart'; // Import the State class from another file

class MyHomePage extends StatefulWidget {

  @override

  MyHomePageState createState() => MyHomePageState();

}

Here ***MyHomePageState*** is our class or widget we are going to create

**Now create a widget and assign it to its state class:**

import 'package:flutter/material.dart';

import 'package:flutter/widgets.dart';

import 'my\_home\_page.dart'; // Import the StatefulWidget if necessary

// State class, now public (no underscore)

class MyHomePageState extends State<MyHomePage> {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        backgroundColor: Colors.cyan,

        title: Text('First Screen'),

      ),

      body: Container(

        width: 100,

        height: 100,

        color: Colors.blue,

        child: Text('A simple Container'),

      ),

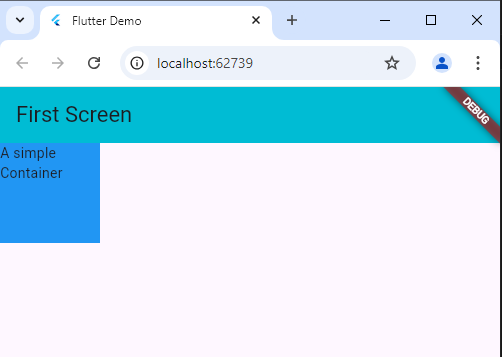
    );

  }

}

* extends State<MyHomePage>: assign it to MyHomePage class.
* Scaffold will give us a canvas to draw any thing on screen so open it.
* Inside scaffold open AppBar and set title and background color.
* After closing appBAr now open body (an area where all of you content will be palced.in body I just open a container (can have only one child). Use its properties and to put content in it I use its child and put text as its child.

**Output:**



**Center widget**

The Center widget in Flutter is a layout widget that centers its child both horizontally and vertically within the available space of its parent.

**Single-Child Layout:** The Center widget can only have one child. If you need to center multiple widgets, you would typically wrap them in a container (e.g., a Column or Row).

**Now I want to make my last created container in center of screen :**

import 'package:flutter/material.dart';

import 'package:flutter/widgets.dart';

import 'my\_home\_page.dart'; // Import the StatefulWidget if necessary

// State class, now public (no underscore)

class MyHomePageState extends State<MyHomePage> {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        backgroundColor: Colors.cyan,

        title: Text('First Screen'),

      ),

      body: Center(

        child: Container(

          width: 100,

          height: 100,

          color: Colors.blue,

          child: Text('A simple Container'),

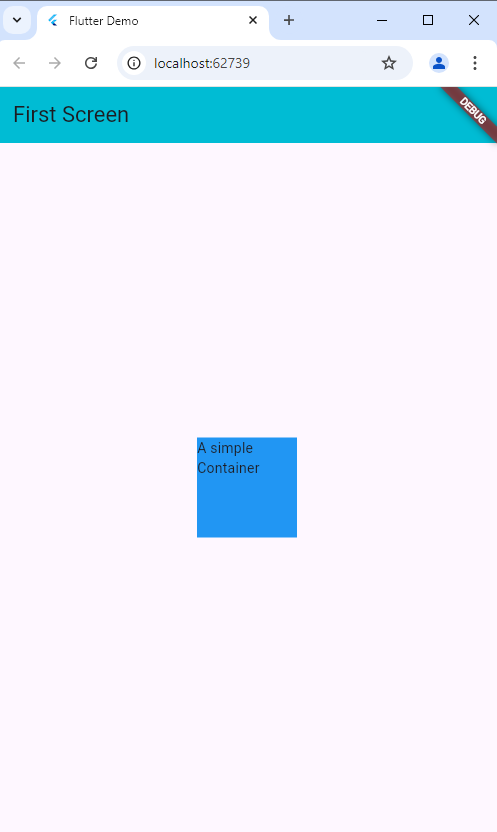
        ),

      ),

    );

  }

}

****

**Now if I want to make text of container in center then?**

*To achieve this, we have to add container as a child of center and then text become child of container because each can have only one child so we have to use this strategy.*

import 'package:flutter/material.dart';

import 'package:flutter/widgets.dart';

import 'my\_home\_page.dart'; // Import the StatefulWidget if necessary

// State class, now public (no underscore)

class MyHomePageState extends State<MyHomePage> {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        backgroundColor: Colors.cyan,

        title: Text('First Screen'),

      ),

      body: Center(

        child: Container(

          width: 100,

          height: 100,

          color: Colors.blue,

          child: Center(child: Text('A simple Container')),

        ),

      ),

    );

  }

}

****

**TEXT in Flutter**

import 'package:flutter/material.dart';

// State class, now public (no underscore)

class Textfile extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        backgroundColor: Colors.cyan,

        title: Text(

  'Flutter Text Styles!',

  style: TextStyle(

    fontSize: 24,             // Font size

    color: Colors.blue,       // Text color

    fontWeight: FontWeight.bold,  // Bold text

    fontStyle: FontStyle.italic,  // Italic text

    letterSpacing: 2.0,       // Space between letters

    wordSpacing: 5.0,         // Space between words

    decoration: TextDecoration.underline, // Underline text

    decorationColor: Colors.red,          // Decoration color

    decorationStyle: TextDecorationStyle.dashed, // Dashed underline

  ),

  textAlign: TextAlign.center,  // Align text to center

  maxLines: 2,  // Maximum number of lines

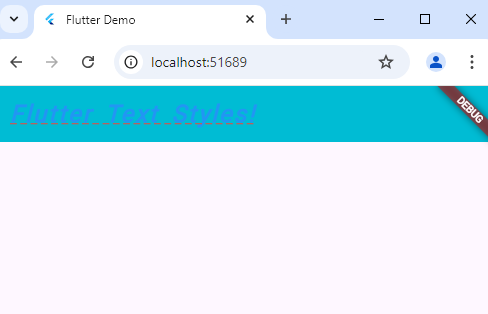
  overflow: TextOverflow.ellipsis,  // Adds "..." when overflowed

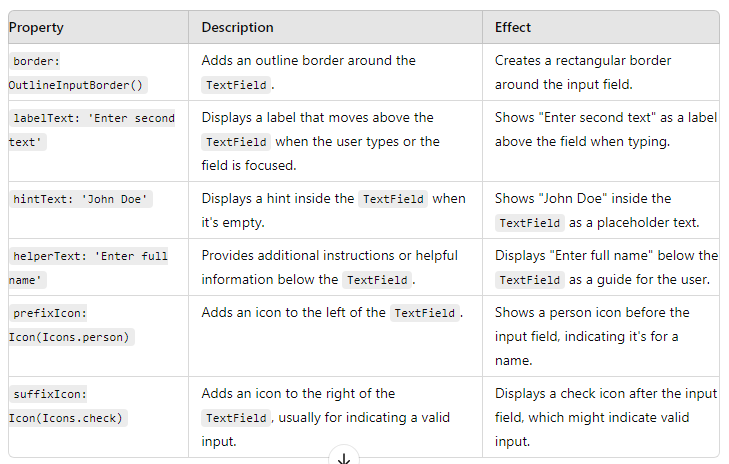
))

    );

  }

}

****



**Buttons in Flutter**

**ElevatedButton:**

* A button that looks raised up with a shadow under it.
* When to use: Use it for important actions that need to stand out, like submitting a form or confirming something.
* "Submit" button when placing an order online.
* "Sign Up" button on a registration form.

**TextButton:**

* A simple button with no background or border. Just text.
* Use it for less important actions like navigation or secondary choices, like "Cancel" or "Forgot Password".
* "Forgot Password" link on a login page.
* "Cancel" button next to a "Submit" button.

**OutlinedButton:**

* A button with a border around it but no background.
* Use it for secondary options, like alternative choices next to more important buttons.
* "Learn More" button next to a "Sign Up" button.
* "No, Thanks" next to a "Subscribe" button.

import 'package:flutter/material.dart';

// State class, now public (no underscore)

class buttons extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

          backgroundColor: Colors.cyan,

          title: Text(

            'Flutter Text Styles!',

          )),

      body: TextButton(

        child: Text('Click Here'),

        onPressed: () {

          print('button clicked');

        },

        onLongPress: () {

          print('Long Pressed clicked');

        },

      ),

    );

  }

}

import 'package:flutter/material.dart';

// State class, now public (no underscore)

class buttons extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

          backgroundColor: Colors.cyan,

          title: Text(

            'Flutter Text Styles!',

          )),

      body: OutlinedButton(

        child: Text('Click Here'),

        onPressed: () {

          print('button clicked');

        },

        onLongPress: () {

          print('Long Pressed clicked');

        },

        style: OutlinedButton.styleFrom(

            backgroundColor: Colors.blue, // Set the background color here

            side: BorderSide(color: Colors.red), // Optional: border color

      ),

    ));

  }

}

import 'package:flutter/material.dart';

// State class, now public (no underscore)

class buttons extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

          backgroundColor: Colors.cyan,

          title: Text(

            'Flutter Text Styles!',

          )),

       body: ElevatedButton(

        onPressed: () {

          print('ElevatedButton clicked');

        },

        child: Text('Click Me'),

        style: ElevatedButton.styleFrom(

          foregroundColor: Colors.white,

          backgroundColor: Colors.blue, // Text color

          elevation: 5,           // Shadow elevation

        ),

      ),

    );

  }

}

**Images in Flutter**

1. Create a folder in your main project and name it assets (you can name it any, but good practice says ‘assets’).
2. Create images folder in assets folder and paste all your project images in that folder.
3. Now go to pubsec.yaml file and do editing.
4. Remove # sign from assets: line and keep it under alignment as following

flutter:

  # The following line ensures that the Material Icons font is

  # included with your application, so that you can use the icons in

  # the material Icons class.

  uses-material-design: true

  # To add assets to your application, add an assets section, like this:

  assets:

    - assets/images/

Remember alignment is so important without proper indenting image will not be loaded.

Now keep an eye on this line.

    - assets/images/

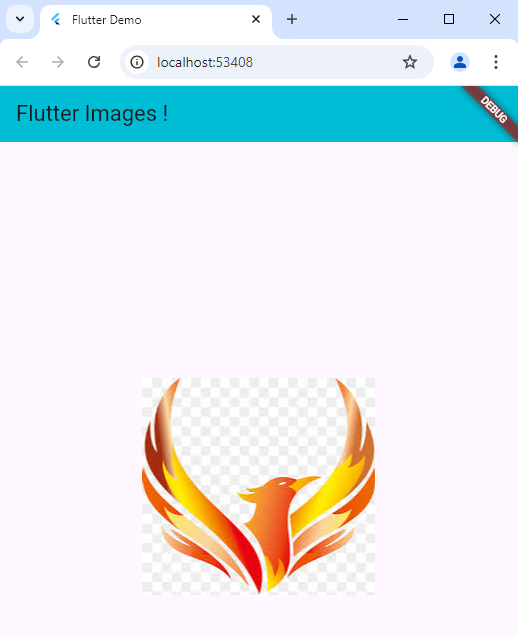
The space between ‘-‘ and assets is not a space but a tab.

The ‘-‘ should be exactly under the 2nd ‘s’ of assets.

1. Now after changes in pubsec.yaml open terminal and write following command:

Flutter pub get

1. After running this command close your project and restart it(not hot reloading).
2. Now go and code of your image.
3. @override
4. Widget build(BuildContext context) {
5. return Scaffold(
6. appBar: AppBar(
7. backgroundColor: Colors.cyan,
8. title: const Text(
9. 'Flutter Images !',
10. )),
11. body:Center(child:Image.asset('assets/images/2.jpg') ,)

****

**Rows & Columns in flutter**

**Row**

A Row is a widget in Flutter that arranges its child widgets in a horizontal line (from left to right). Use a Row when you want to display items side by side.

**Example**

Imagine you want to display three buttons side by side: "Home", "Profile", and "Settings".(like in fb).

**Column**

A Column is a widget that arranges its child widgets in a vertical line (from top to bottom).Use a Column when you want to display items one on top of the other.

**Example**

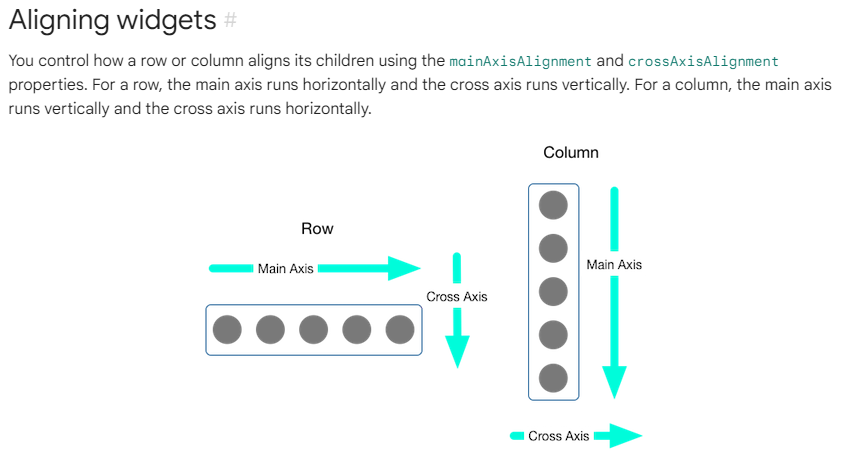
Now, let's say you want to display a title at the top and a subtitle below it.

**Expanded**

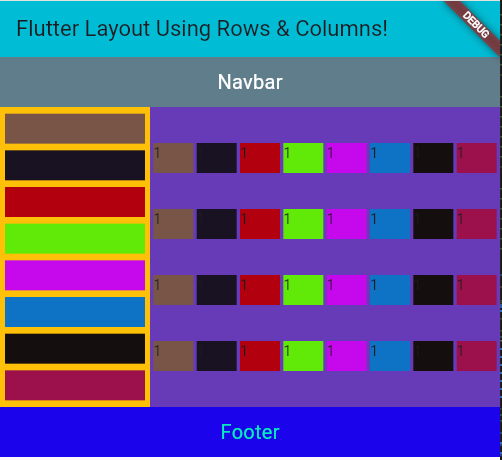
The Expanded widget allows a child widget to take up all available space within a Row or Column. It tells Flutter to expand the child to fill the remaining space.Use Expanded when you want one widget to take up more space than others in a Row or Column.

**Example**

In a chat application, the screen layout often includes a list of messages above and an input field at the bottom. The message list can be wrapped in an Expanded widget to ensure it takes all the available space, while the input field stays at the bottom with a fixed height.

****

***To understand the all above concept properly we are going to create a following Layout:***

****

The layout consists of a scaffold that includes an app bar with the title "Flutter Layout Using Rows & Columns!" and a body divided into a vertical column.

* At the top, there's a fixed-height navigation bar, followed by a Row that contains two main sections: a left sidebar and a right expanded area.
* The left sidebar, with a fixed width of 150 pixels and height of 300 pixels, contains a vertical list of colored containers arranged in a Column, each serving as individual elements.
* The right section, wrapped in an Expanded widget, also has a fixed height of 300 pixels and consists of multiple rows displaying smaller colored containers with text labels.
* Finally, there is a footer at the bottom of the screen, also with a fixed height, completing the layout.

**Let’s code step by step.**

1. **App Bar**

class row\_column extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        backgroundColor: Colors.cyan,

        title: const Text('Flutter Layout Using Rows & Columns!'),

      ),

1. **Body :**

We wrap whole body into column the reason is we need stack like after navbar we have aside and section and then footer which are place top and bottom of each other.

Now in our column our whole-body layout resides. Now all thing will become its children so add property of children first .

 body: Column(

        children: [];

Now in children array first child I need is navbar so add properties:

Container(

            color: Colors.blueGrey,

            height: 50, // Fixed height for the navbar

            width: double.infinity, // Full width of the screen

            child: Center(

              child: Text(

                'Navbar',

                style: TextStyle(color: Colors.white, fontSize: 20),

              ),

            ),

          ),

After navbar we need a row because we have to arrange aside and section horizontally. This row become child of our column ad then it will have its own child.

Row(

            children: [

              Container(

                width: 150,

                height: 300, // Fixed height for the containers

                color: Colors.amber,

the first child container will draw an aside. Now 2nd child of tis row is section which will take whole remaining space.we use Expanded for this purpose as we learned above expanded is use to contain rest of space.no need to assign any width here.

Expanded(

                child: Container(

                  height: 300, // Keep the height at 300

                  color: Colors.deepPurple,

now till this step we have created aside a section now I want to add some items in aside vertically as shown in picture above. We created a container for aside which can only have one child so we call his child as column, because we want multiple items vertically. Column has multiple children also have vertical decoration.

child: Column(

mainAxisAlignment: MainAxisAlignment.spaceEvenly,

                  children: [

                    Container(width: 140, height: 30, color: Colors.brown),

                    Container(

                        width: 140,

                        height: 30,

                        color: Color.fromARGB(255, 25, 18, 34)),

                    Container(

                        width: 140,

                        height: 30,

                        color: Color.fromARGB(255, 179, 0, 15)),

                    Container(

                        width: 140,

                        height: 30,

                        color: Color.fromARGB(255, 97, 233, 7)),

                    Container(

                        width: 140,

                        height: 30,

                        color: Color.fromARGB(255, 198, 8, 236)),

                    Container(

                        width: 140,

                        height: 30,

                        color: Color.fromARGB(255, 14, 115, 197)),

                    Container(

                        width: 140,

                        height: 30,

                        color: Color.fromARGB(255, 20, 14, 14)),

                    Container(

                        width: 140,

                        height: 30,

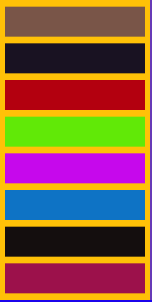
                        color: Color.fromARGB(255, 156, 17, 75)),

                  ],

                ),

              ),

Now column has multiple children, we add all. Next step is to create space between each child of column so we used mainAxisAlignment: MainAxisAlignment.spaceEvenly, to add space vertically. Here our aside task has been done.



**Now move toward section:**

In section we have 8 columns and 4 rows having containers.it is become 2d so we have to look it up in both ways column wise and row wise. First, I am taking column and in that column I will put each row so they will stack.

Expanded(

                child: Container(

                  height: 300, // Keep the height at 300

                  color: Colors.deepPurple,

                  child: Column(

                    mainAxisAlignment: MainAxisAlignment.spaceEvenly,

                    children: [

                      Row(

                        mainAxisAlignment: MainAxisAlignment.spaceEvenly,

                        children: [

                          Container(

                            width: 40,

                            height: 30,

                            color: Colors.brown,

                            child: Text('1'),

                          ),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 25, 18, 34),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 179, 0, 15),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 97, 233, 7),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 198, 8, 236),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 14, 115, 197),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 20, 14, 14),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 156, 17, 75),

                              child: Text('1')),

                        ],

                      ),

                      Row(

                        mainAxisAlignment: MainAxisAlignment.spaceEvenly,

                        children: [

                          Container(

                            width: 40,

                            height: 30,

                            color: Colors.brown,

                            child: Text('1'),

                          ),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 25, 18, 34),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 179, 0, 15),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 97, 233, 7),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 198, 8, 236),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 14, 115, 197),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 20, 14, 14),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 156, 17, 75),

                              child: Text('1')),

                        ],

                      ),

                      Row(

                        mainAxisAlignment: MainAxisAlignment.spaceEvenly,

                        children: [

                          Container(

                            width: 40,

                            height: 30,

                            color: Colors.brown,

                            child: Text('1'),

                          ),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 25, 18, 34),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 179, 0, 15),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 97, 233, 7),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 198, 8, 236),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 14, 115, 197),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 20, 14, 14),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 156, 17, 75),

                              child: Text('1')),

                        ],

                      ),

                      Row(

                        mainAxisAlignment: MainAxisAlignment.spaceEvenly,

                        children: [

                          Container(

                            width: 40,

                            height: 30,

                            color: Colors.brown,

                            child: Text('1'),

                          ),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 25, 18, 34),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 179, 0, 15),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 97, 233, 7),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 198, 8, 236),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 14, 115, 197),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 20, 14, 14),

                              child: Text('1')),

                          Container(

                              width: 40,

                              height: 30,

                              color: Color.fromARGB(255, 156, 17, 75),

                              child: Text('1')),

                        ],

                      ),

                    ],

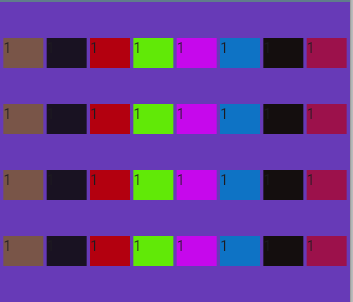
                  ),

                ),

              ),

            ],

          ),

****

Now the last part is footer which is a simple container

 Container(

            color: Color.fromARGB(255, 27, 4, 236),

            height: 50, // Fixed height for the navbar

            width: double.infinity, // Full width of the screen

            child: Center(

              child: Text(

                'Footer',

                style: TextStyle(color: Color.fromARGB(255, 5, 238, 187), fontSize: 20),

              ),

            ),

          ),

        ],

      ),

    );

  }

}

****

**Inkwell in flutter (make clickable any widget)**

Imagine you have a picture or a box in your app, and you want something to happen when you tap on it (like opening a new page or showing a message). By wrapping that widget with Inkwell, it becomes tappable and shows a ripple (splash) effect when touched. it can be applied to almost any widget.

**Wrap the whole widget which you want to make clickable into inkwell widget.**

import 'package:flutter/material.dart';

// State class, now public (no underscore)

class InkWellWidget  extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        backgroundColor: Colors.cyan,

        title: const Text('First Screen'),

      ),

      body: InkWell(

        onTap: (){

          print('container is tapped');

        },

        onDoubleTap:  (){

          print('container is doubletapped');

        },

        onLongPress:  (){

          print('container is longtapped');

        },

        child: Center(

          child: Container(

            width: 200,

            height: 200,

            color: Colors.blue,

            child: const Center(child: Text('A simple Container')),

          ),

        ),

      ),

    );

  }

}

**With Image:**

import 'package:flutter/material.dart';

// StatelessWidget with multiple clickable containers

class InkWellWidget extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        backgroundColor: Colors.cyan,

        title: const Text('First Screen'),

      ),

      body: Column(

        mainAxisAlignment:

            MainAxisAlignment.center, // Center the containers vertically

        children: [

          // First clickable container (original)

          InkWell(

            onTap: () {

              print('First container (with image) is tapped');

            },

            onDoubleTap: () {

              print('First container (with image) is double-tapped');

            },

            onLongPress: () {

              print('First container (with image) is long-pressed');

            },

            child: Center(

              child: Container(

                width: 200,

                height: 200,

                color: Colors.blue,

                child: Center(

                  child: Image.asset('assets/images/2.jpg',

                   // Example image URL

                    fit: BoxFit.cover,

                  ),

                ),

              ),

            ),

          ),

          SizedBox(height: 20), // Space between the two containers

          // Second clickable container (new)

          InkWell(

            onTap: () {

              print('Second container  is tapped');

            },

            onDoubleTap: () {

              print('Second container  is double-tapped');

            },

            onLongPress: () {

              print('Second container is long-pressed');

            },

            child: Center(

              child: Container(

                width: 200,

                height: 200,

                color: Colors.green,

                child: Center(

                  child: Text('A Simple Container')

                ),

              ),

            ),

          ),

        ],

      ),

    );

  }

}

**ScrollView**

In Flutter, a ScrollView is used to create scrollable content when you have widgets that overflow the screen. The most commonly used widget for scrolling in Flutter is the SingleChildScrollView, which allows a single child to be scrollable in either the vertical or horizontal direction. If you have multiple children, you can use ListView, GridView, or other specialized scrolling widgets.

**Column Wise (Vertical Scroll)**

import 'package:flutter/material.dart';

// StatelessWidget with multiple clickable containers

class ScrollViewWidget extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        backgroundColor: Colors.cyan,

        title: const Text('First Screen'),

      ),

      body: SingleChildScrollView(

        child: Column(

          children: [

            Center(

              child: Container(

                height: 150,

                color: Colors.greenAccent,

              ),

            ),

            SizedBox(height: 20),

            Center(

              child: Container(

                height: 150,

                color: Color.fromARGB(255, 91, 2, 233),

              ),

            ),

            SizedBox(height: 20),

            Center(

              child: Container(

                height: 150,

                color: Color.fromARGB(255, 7, 228, 25),

              ),

            ),

            SizedBox(height: 20),

            Center(

              child: Container(

                height: 150,

                color: Color.fromARGB(255, 228, 9, 38),

              ),

            ),

            SizedBox(height: 20),

            Center(

              child: Container(

                height: 150,

                color: Color.fromARGB(255, 1, 35, 63),

              ),

            ),

            SizedBox(height: 20),

            Center(

              child: Container(

                height: 150,

                color: Color.fromARGB(255, 252, 3, 148),

              ),

            ),

            SizedBox(height: 20),

            Center(

              child: Container(

                height: 150,

                color: Colors.orangeAccent,

              ),

            ),

          ],

        ),

      ),

    );

  }

}

**Row Wise (horizontally Scroll)**

Note: chrome will not allow horizontally scrolling through mouse.so go to my app and add line for mouse scrolling horizontally.

Caution: only those elements will be scrolled who are in ScrollView widget.other part will not be.

 @override

  Widget build(BuildContext context) {

    return MaterialApp(

      title: 'Flutter Demo',

      home: ScrollViewWidget(),

*scrollBehavior: const MaterialScrollBehavior().copyWith(*

*dragDevices: {PointerDeviceKind.mouse},)*

    );

  }

import 'package:flutter/material.dart';

// StatelessWidget with horizontal scrollable containers

class ScrollViewWidget extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        backgroundColor: Colors.cyan,

        title: const Text('First Screen'),

      ),

      body: SingleChildScrollView(

        scrollDirection: Axis.horizontal,

        child: Row(

          children: [

            Container(

              height: 100,

              width: 100,

              color: Colors.grey,

            ),

            Container(

              height: 100,

              width: 100,

              color: const Color.fromARGB(255, 240, 20, 20),

            ),

            Container(

              height: 100,

              width: 100,

              color: const Color.fromARGB(255, 15, 13, 13),

            ),

            Container(

              height: 100,

              width: 100,

              color: const Color.fromARGB(255, 47, 25, 128),

            ),

            Container(

              height: 100,

              width: 100,

              color: const Color.fromARGB(255, 130, 241, 3),

            ),

            Container(

              height: 100,

              width: 100,

              color: const Color.fromARGB(255, 75, 3, 243),

            ),

            Container(

              height: 100,

              width: 100,

              color: const Color.fromARGB(255, 31, 158, 243),

            ),

            Container(

              height: 100,

              width: 100,

              color: const Color.fromARGB(255, 236, 128, 4),

            ),

            Container(

              height: 100,

              width: 100,

              color: const Color.fromARGB(255, 104, 1, 1),

            ),

          ],

        ),

      ),

    );

  }

}

**Margin and Padding**

**Padding:** Adds space inside a widget, between the widget's content and its border.

**Margin:** Adds space outside a widget, between the widget and other widgets around it.

****

import 'package:flutter/material.dart';

// StatelessWidget with horizontal scrollable containers

class MarginPadding extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

        appBar: AppBar(

          backgroundColor: Colors.cyan,

          title: const Text('First Screen'),

        ),

        body: Center(

          child: Row(

            children: [

              Container(

                width: 100,

                height: 100,

                color: Colors.purple,

                child: Text('Container 1'),

              ),

              Container(

                width: 100,

                height: 100,

                color: Color.fromARGB(255, 85, 2, 240),

                child: Text('Container 2'),

              )

            ],

          ),

        ));

  }

}

As we can see there are two containers with some text now, I want to move text of 1st container.for this purpose we have to do padding of our text.

**Padding Parameters:**

1. **EdgeInsets.all(double value):**

* This applies the same amount of padding on all four sides (top, bottom, left, and right).

1. **EdgeInsets.symmetric({double vertical, double horizontal}):**

* This applies different padding for vertical (top and bottom) and horizontal (left and right) sides.
* Example: Adds 10 pixels of padding on the top and bottom, and 20 pixels on the left and right.

1. **EdgeInsets.only({double left, double top, double right, double bottom}):**

* This allows you to specify padding for each side individually.
* Example: Adds 10 pixels of padding on the left, 20 on the top, 30 on the right, and 40 on the bottom.

import 'package:flutter/material.dart';

class MarginPadding extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

        appBar: AppBar(

          backgroundColor: Colors.cyan,

          title: const Text('First Screen'),

        ),

        body: Center(

          child: Row(

            children: [

              Container(

                width: 100,

                height: 100,

                color: Colors.purple,

                child: Padding(

                  padding: const EdgeInsets.only(left: 04, top: 35, right: 15, bottom: 20),

                  child: Text('Container 1'),

                ),

              ),

              Container(

                width: 100,

                height: 100,

                color: Color.fromARGB(255, 85, 2, 240),

                child: Padding(

                  padding: const EdgeInsets.all(8.0),

                  child: Text('Container 2'),

                ),

              )

            ],

          ),

        ));

  }

}

****

**Margin Parameters:**

The margin parameter is similar to padding, but it adds space outside a widget, between the widget itself and other surrounding widgets. Like padding, the margin is also defined using the EdgeInsets class.

Margins can be applied to widgets like Container, Card, and any widget that accepts margin as a parameter.

import 'package:flutter/material.dart';

// StatelessWidget with horizontal scrollable containers

class MarginPadding extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

        appBar: AppBar(

          backgroundColor: Colors.cyan,

          title: const Text('First Screen'),

        ),

        body: Center(

          child: Row(

            children: [

              Container(

                width: 100,

                height: 100,

                color: Colors.purple,

                margin: EdgeInsets.only(left: 04, top: 35, right: 15, bottom: 20),

                child: Padding(

                  padding: const EdgeInsets.only(left: 04, top: 35, right: 15, bottom: 20),

                  child: Text('Container 1'),

                ),

              ),

              Container(

                width: 100,

                height: 100,

                color: Color.fromARGB(255, 85, 2, 240),

                margin: EdgeInsets.all(35),

                child: Padding(

                  padding: const EdgeInsets.all(8.0),

                  child: Text('Container 2'),

                ),

              )

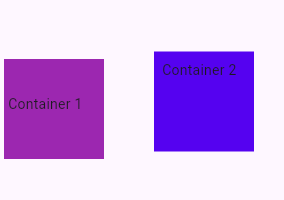
            ],

          ),

        ));

  }

}

****

**BOXDECORATION**

The BoxDecoration class is used to decorate a container, allowing you to apply things like background colors, borders, gradients, shadows, and even rounded corners.

You can apply BoxDecoration to a Container through its decoration parameter.

**BoxDecoration Properties:**

**color:** Background color of the container.

**borderRadius:** Rounds the corners of the container.

**border:** Adds a border around the container.

**boxShadow:** Adds shadow around the container.

**gradient:** Applies a color gradient inside the container.

**image:** Adds an image as the background of the container.

import 'package:flutter/material.dart';

// StatelessWidget with horizontal scrollable containers

class BocDecorationClass extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

        appBar: AppBar(

          backgroundColor: Colors.cyan,

          title: const Text('First Screen'),

        ),

        body: Center(

        child: Container(

          width: 200,

          height: 200,

          decoration: BoxDecoration(

            color: Colors.blue, // Background color

            borderRadius: BorderRadius.only(topLeft:Radius.circular(20),bottomRight:Radius.circular(20)), // Rounded corners

            border: Border.all(

              color: Colors.black,

              width: 3, // Border width

            ),

            boxShadow: [

              BoxShadow(

                color: Colors.black26, // Shadow color

                // Shadow position

                blurRadius: 11,

                spreadRadius: 23 // How much the shadow spreads

              ),

            ],

            gradient: LinearGradient(

              colors: [Colors.red, Colors.yellow,Colors.blue,Colors.black], // Gradient colors can add as many as you want

              begin: Alignment.topLeft,

              end: Alignment.bottomRight,

            ),

          ),

          child: Center(

            child: Text(

              'Styled Container',

              style: TextStyle(color: Colors.black, fontSize: 16),

            ),

          ),

        ),

      ),

    );

  }

}

****

**ListView In Flutter**

A ListView in Flutter is a scrollable list of widgets. It allows users to scroll vertically (or horizontally) through a list of items that can either be a fixed number of widgets or dynamically loaded as the user scrolls (for large datasets).

There are several types of ListView that are commonly used, including:

* ListView (default constructor) — for a list of widgets.
* ListView.builder — for building a list dynamically.
* ListView.separated — for building a list with separators between items.
* ListView.custom — for creating a custom list view.

**ListView (default constructor):**

import 'package:flutter/material.dart';

// StatelessWidget with horizontal scrollable containers

class Listviews extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

        appBar: AppBar(

          backgroundColor: Colors.cyan,

          title: const Text('First Screen'),

        ),

        body: ListView(

          children: [

            Padding(

              padding: const EdgeInsets.all(20.0),

              child: Text('Owais Khan'),

            ),

            Padding(

              padding: const EdgeInsets.all(20.0),

              child: Text('Owais Khan'),

            ),

            Padding(

              padding: const EdgeInsets.all(20.0),

              child: Text('Owais Khan'),

            ),

            Padding(

              padding: const EdgeInsets.all(20.0),

              child: Text('Owais Khan'),

            ),

            Padding(

              padding: const EdgeInsets.all(20.0),

              child: Text('Owais Khan'),

            ),

          ],

        ));

  }

}

**For horizontally scrolling :**

body: ListView(

           scrollDirection: Axis.horizontal,

**ListView.builder — for building a list dynamically.**

import 'package:flutter/material.dart';

class ListViewBuilderExample extends StatelessWidget {

  var myarr = [

    'Ahmed', 'Ali', 'Sara', 'Hina', 'Zain', 'Arsalan', 'Ayesha', 'Farah', 'Kashif', 'Rida',

    'Bilal', 'Samina', 'Adeel', 'Mehwish', 'Asim', 'Sana', 'Hassan', 'Zoya', 'Tariq', 'Faisal',

    'Nida', 'Imran', 'Fatima', 'Saad', 'Nawaz', 'Yasmin', 'Shahid', 'Taimoor', 'Iqra', 'Hamza',

    'Arifa', 'Shazia', 'Usman', 'Jawad', 'Bushra', 'Amna', 'Murtaza', 'Lubna', 'Rashid', 'Saima',

    'Kiran', 'Zeeshan', 'Neha', 'Salman', 'Waqas', 'Mansoor', 'Faiza', 'Raheel', 'Sadia', 'Arham'

  ];

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(title: Text('ListView.builder')),

      body: ListView.builder(itemBuilder: (context,index){

        return Text(myarr[index]);

      },

      itemCount: myarr.length,

      itemExtent: 50,

      scrollDirection: Axis.horizontal,

      )

    );

  }

}

**ListView.builder**:

* This is a special constructor for creating a scrollable list where the items are built lazily as they scroll into view.
* Unlike a regular ListView where all items are created at once, ListView.builder only builds the widgets that are currently visible on the screen, making it more efficient for long lists.

**itemBuilder:**

* itemBuilder is a required parameter in ListView.builder. It is a function that tells Flutter how to build each item in the list.
* The itemBuilder function is called every time Flutter needs to display a list item, whether for the first time or when scrolling new items into view.

**(context,index) :** *these are not keywords you can pass any two variable name .*

* context tells Flutter where this widget is located and it helps in building the widget properly.
* index is **just a number** that tells you the **position of an item** in the list. It starts from **0** by default and increases by 1 for each item.
* itemExtent defines the height (for vertical lists) or width (for horizontal lists) of each item.

**ListView.separated — for building a list with separators between items.**

import 'package:flutter/material.dart';

class ListViewSeparatedExample extends StatelessWidget {

  var myarr = [

    'Ahmed', 'Ali', 'Sara', 'Hina', 'Zain', 'Arsalan', 'Ayesha', 'Farah', 'Kashif', 'Rida',

    'Bilal', 'Samina', 'Adeel', 'Mehwish', 'Asim', 'Sana', 'Hassan', 'Zoya', 'Tariq', 'Faisal',

    'Nida', 'Imran', 'Fatima', 'Saad', 'Nawaz', 'Yasmin', 'Shahid', 'Taimoor', 'Iqra', 'Hamza',

    'Arifa', 'Shazia', 'Usman', 'Jawad', 'Bushra', 'Amna', 'Murtaza', 'Lubna', 'Rashid', 'Saima',

    'Kiran', 'Zeeshan', 'Neha', 'Salman', 'Waqas', 'Mansoor', 'Faiza', 'Raheel', 'Sadia', 'Arham'

  ];

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(title: Text('ListView.separated Example')),

      body: ListView.separated(

        itemBuilder: (context, index) {

          return Padding(  // Added padding to make items more visible

            padding: const EdgeInsets.all(16.0),

            child: Text(

              myarr[index],

              style: TextStyle(fontSize: 18), // Increase font size for better visibility

            ),

          );

        },

        itemCount: myarr.length,

        separatorBuilder: (context, index) {

          return Divider(

            height: 30,  // Space between text and the divider

            thickness: 2,  // Thickness of the divider

            color: Colors.grey,  // Divider color

          );

        },

      ),

The separatorBuilder is a callback function that Flutter calls for each item to determine what should be displayed between the list items. Just like itemBuilder is responsible for building the list items, separatorBuilder is responsible for creating the widget that separates those items.

**ListTile in Flutter**

The ListTile widget is commonly used to display a row with a title, subtitle, and leading/trailing widgets.

**ListTile Components:**

**leading:** A widget placed at the start of the ListTile, usually used for icons, avatars, or any other widget you want to display on the left side.

**title:** The main text of the tile, typically the name or main content.

**subtitle:** An optional secondary text that appears below the title (like additional information).

**trailing:** A widget displayed at the end of the tile, usually an icon or button, often used for actions like navigation or deletion.

**onTap:** The function that is called when the ListTile is tapped (used to add interactivity).

import 'package:flutter/material.dart';

class ListTileExample extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('ListTile with Trailing & onTap'),

      ),

      body: ListView.separated(

        itemCount: 6, // Number of ListTiles

        itemBuilder: (context, index) {

          return ListTile(

            leading: Icon(Icons.person), // Icon at the start

            title: Text('User ${index + 1}'), // Dynamic user name

            subtitle: Text('Subtitle for User ${index + 1}'), // Dynamic subtitle

            trailing: Icon(Icons.arrow\_forward\_ios), // Icon at the end (trailing)

            // Action when the ListTile is tapped

            onTap: () {

              print('Tapped on User ${index + 1}');

            },

          );

        },

        separatorBuilder: (context, index) {

          return Divider(

            thickness: 1.0,

            color: Colors.grey,

          );

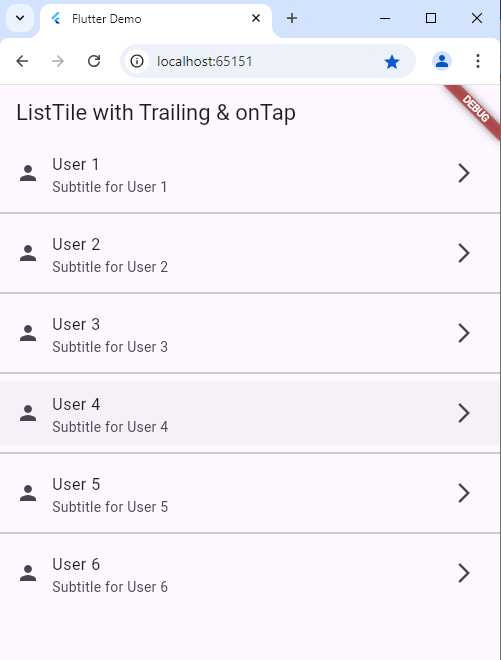
        },

      ),

    );

  }

}



**CircleAvatar**

**CircleAvatar** is a handy widget used to display circular images, icons, or text, often used to represent a user profile picture, an avatar, or any other type of circular content.

**Properties of CircleAvatar:**

**radius:**

* Defines the size (radius) of the avatar (the circle).
* Example: radius: 50 will make the avatar bigger.

**backgroundColor:**

* Sets the background color of the avatar if no image is provided.
* Example: backgroundColor: Colors.blue.

**backgroundImage:**

* This property is used to set an image inside the circular avatar.
* It takes a NetworkImage (for loading an image from the internet) or an AssetImage (for loading from your local assets).

**child:**

* If you want to display an icon or text inside the avatar, you can use the child property.

**maxRadius:** Specifies the maximum size of the avatar.

**minRadius:** Specifies the minimum size of the avatar.

These properties give you finer control over the size when it's dynamically resized

import 'package:flutter/material.dart';

class AvatarExample extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('Avatar Examples'),

      ),

      body: Center(

        child: Column(

          mainAxisAlignment: MainAxisAlignment.spaceEvenly,

          children: [

            CircleAvatar(

              backgroundColor: Colors.blue,

              child: Text('A', style: TextStyle(fontSize: 24, color: Colors.white)),

            ),

            CircleAvatar(

              radius: 50,  // Larger size

              backgroundImage: AssetImage('assets/images/2.jpg'),

            ),

            CircleAvatar(

              radius: 40,

              child: Icon(Icons.person, size: 40),

              backgroundColor: Colors.green,

            ),

          ],

        ),

      ),

    );

  }

}

**Avatar with border**

CircleAvatar does not have a built-in border property, wrapping it in a Container with a circular shape and decoration allows you to add a border around the avatar.

import 'package:flutter/material.dart';

class AvatarBorder extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

        appBar: AppBar(

          title: Text('Avatar Examples'),

        ),

        body: Column(

          children: [

            Container(

              margin: EdgeInsets.all(60),//to move circle away from appbar

              decoration: BoxDecoration(

                shape: BoxShape.circle, // Makes the Container circular

                border: Border.all(

                  color: Color.fromARGB(255, 144, 253, 2), // Border color

                  width: 2.0,

                  // Border width

                ),

                boxShadow: [

                  BoxShadow(

                      color: Colors.black26, // Shadow color

                      // Shadow position

                      blurRadius: 11,

                      spreadRadius: 23 // How much the shadow spreads

                      ),

                ],

              ),

              child: CircleAvatar(

                radius: 50, // Size of the avatar

                backgroundImage: AssetImage('assets/images/2.jpg'),

              ),

            )

          ],

        ));

  }

}

**Avatar List With ListTile**

import 'package:flutter/material.dart';

class AvatarListViewSeparated extends StatelessWidget {

  // Static array of names

  final List<String> myarr = [

    'Ahmed', 'Ali', 'Sara', 'Hina', 'Zain', 'Arsalan', 'Ayesha', 'Farah', 'Kashif', 'Rida',

    'Bilal', 'Samina', 'Adeel', 'Mehwish', 'Asim', 'Sana', 'Hassan', 'Zoya', 'Tariq', 'Faisal',

    'Nida', 'Imran', 'Fatima', 'Saad', 'Nawaz', 'Yasmin', 'Shahid', 'Taimoor', 'Iqra', 'Hamza',

    'Arifa', 'Shazia', 'Usman', 'Jawad', 'Bushra', 'Amna', 'Murtaza', 'Lubna', 'Rashid', 'Saima',

    'Kiran', 'Zeeshan', 'Neha', 'Salman', 'Waqas', 'Mansoor', 'Faiza', 'Raheel', 'Sadia', 'Arham'

  ];

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('ListView with Avatars'),

      ),

      body: ListView.separated(

        itemCount: myarr.length, // Number of items

        separatorBuilder: (context, index) => Divider(), // Separator between items

        itemBuilder: (context, index) {

          // Get the current name

          String name = myarr[index];

          // Get the first two letters for the CircleAvatar

          String initials = name.substring(0, 2).toUpperCase();

          return ListTile(

            leading: CircleAvatar(

              radius: 25, // Adjust the size of the avatar

              backgroundColor: Colors.blue, // Avatar background color

              child: Text(

                initials, // Display first two letters

                style: TextStyle(

                  color: Colors.white,

                  fontWeight: FontWeight.bold,

                ),

              ),

            ),

            title: Text(name), // Full name as title

            subtitle: Text('Subtitle for $name'),

            trailing: Icon(Icons.arrow\_forward\_ios),

          );

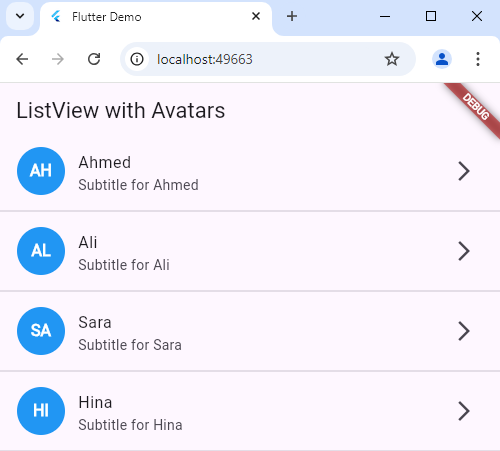
        },

      ),

    );

  }

}

****

**Flutter Card Widget**

The Card widget in Flutter is a material design widget that provides a container with rounded corners, shadows, and a background color. It's commonly used to display related information, such as contact info, product details, or images, in a card-like UI.

* Rounded corners and elevation (shadows).
* Can contain any widget like Text, Image, or ListTile.
* Often used for presenting information that is conceptually related

**Properties of Card:**

**color:** Sets the background color of the card.

**margin:** Adds margin around the card.

**shape:** Allows customization of the card’s shape, such as rounded corners.

**elevation:** Adjusts the shadow depth.

**shadowColor:** for shadow color.

import 'package:flutter/material.dart';

class CardExample extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('Flutter Card Example'),

      ),

      body: Padding(

        padding: const EdgeInsets.all(16.0),

        child: Column(

          children: [

            Card(

              elevation: 12, // Shadow depth of the card

              color: Colors.red,

              shadowColor: Colors.amber,

              shape: RoundedRectangleBorder(

                borderRadius: BorderRadius.circular(15), // Rounded corners

              ),

              child: Padding(

                padding: const EdgeInsets.all(16.0),

                child: Column(

                  crossAxisAlignment: CrossAxisAlignment.start,

                  children: [

                    Text(

                      'Card Title',

                      style: TextStyle(

                        fontSize: 20,

                        fontWeight: FontWeight.bold,

                      ),

                    ),

                    Text(

                      'This is a simple card description. You can add text or any other widget inside a card.',

                    ),

                  ],

                ),

              ),

            ),

            SizedBox(height: 10),

            Card(

              elevation: 12, // Shadow depth of the card

              color: Color.fromARGB(255, 4, 227, 243),

              shadowColor: Color.fromARGB(255, 4, 7, 150),

              shape: RoundedRectangleBorder(

                borderRadius: BorderRadius.circular(15), // Rounded corners

              ),

              child: Padding(

                padding: const EdgeInsets.all(16.0),

                child: ListTile(

                  leading: CircleAvatar(

                      radius: 55, // Adjust the size of the avatar

                      backgroundColor: Color.fromARGB(255, 171, 241, 6), // Avatar background color

                      backgroundImage: AssetImage('assets/images/1.jpg'),

                      ),

                  title: Text('Card with Avatar'), // Full name as title

                  subtitle: Text('Subtitle for Card  with Avatar'),

                  trailing: Icon(Icons.arrow\_forward\_ios),

                ),

              ),

            ),

          ],

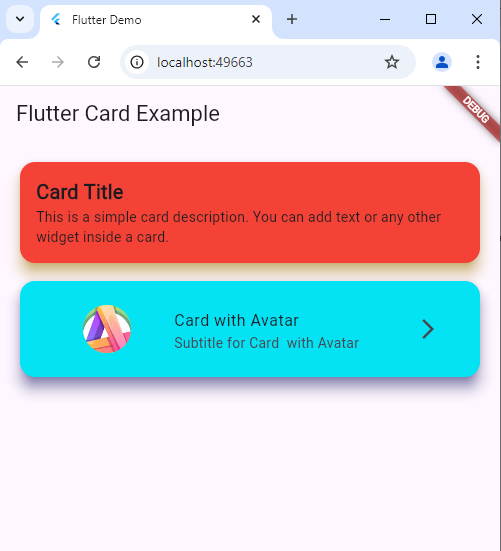
        ),

      ),

    );

  }

}

****

**GridView in Flutter**

GridView is a widget in Flutter used to display a list of items in a grid format, which is perfect when you want multiple items displayed in a structured and evenly distributed layout, like image galleries, product lists in e-commerce apps, etc.

**Types of GridView**

Flutter provides four types of GridView constructors:

1. **GridView.builder** – Best for large data, loads items lazily.
2. **GridView.count** – For creating a grid with a fixed number of items per row.
3. **GridView.extent** – Allows specifying the maximum cross-axis (column) extent.
4. **GridView.custom** – Provides full control over the grid’s children and layout.

**crossAxisCount:**

The property crossAxisCount is used when you're defining the layout of a grid in Flutter. It's one of the most essential properties to control how many items you want to display horizontally across the grid's cross-axis (columns). It defines how many items (widgets) you want in each row of your grid.

***Main Axis:*** This is the scroll direction of your grid (usually vertical by default). If you're scrolling vertically, the main axis is vertical.

***Cross Axis:*** This is perpendicular to the scroll direction. If the grid scrolls vertically, the cross-axis is horizontal, and if the grid scrolls horizontally, the cross-axis is vertical.

**How crossAxisCount Works**

Fixed Number of Items per Row: crossAxisCount defines a fixed number of items in each row. For example:

* If crossAxisCount is set to 2, it means 2 items per row in a vertically scrolling grid (2 columns).
* If crossAxisCount is set to 3, it means 3 items per row in a vertically scrolling grid (3 columns).
* Responsive Grid Layout: This property makes it easy to create responsive grid layouts. Based on the available screen width, you can decide how many items should fit in a row.

import 'package:flutter/material.dart';

class GridExample extends StatelessWidget {

    final List<String> imageUrls = [

    'assets/images/2.jpg',

    'assets/images/1.jpg',

    'assets/images/3.png',

    'assets/images/6.png',

    'assets/images/4.png',

    'assets/images/3.png',

  ];

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('Grid Examples'),

      ),

      body: GridView.builder(

      padding: const EdgeInsets.all(10.0),

      itemCount: imageUrls.length,

      gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(

        crossAxisCount: 2, // 2 items per row

        crossAxisSpacing: 10.0, // Spacing between columns

        mainAxisSpacing: 10.0, // Spacing between rows

childAspectRatio: 1.0,

      ),

      itemBuilder: (context, index) {

        return Image.asset(imageUrls[index]); // Load image from URL

      },

    ),

    );

  }

}

The childAspectRatio controls the shape of each grid item. By default, a value of 1.0 makes items square (width = height). You can adjust this value to create different aspect ratios.

**Example 1: Wider Items (Aspect Ratio 2:1)**

* If you want the items to be wider than they are tall, set childAspectRatio to a value greater than 1.0, such as 2.0:
* In this case, each item will have a width twice its height, making them rectangular and stretched horizontally.

**Example 2: Taller Items (Aspect Ratio 0.5)**

* If you want the items to be taller than they are wide, set childAspectRatio to a value less than 1.0, such as 0.5:
* the items will have a height that is twice their width, making them taller rectangles.

**When to Use SliverGridDelegateWithFixedCrossAxisCount**

You should use SliverGridDelegateWithFixedCrossAxisCount when:

* You want to control the exact number of items per row or column.
* You want fixed-sized grid items across different screen sizes.
* You need a consistent grid structure where the number of columns is known ahead of time (e.g., in an image gallery or product display).

****

**Border Around Each Item**

To add a border and shadow around each item in your grid, you need to wrap the Image.asset widget inside a Container and apply both a border and a box shadow using the BoxDecoration widget.

import 'package:flutter/material.dart';

class GridWithBorder extends StatelessWidget {

  final List<String> imageUrls = [

    'assets/images/2.jpg',

    'assets/images/1.jpg',

    'assets/images/3.png',

    'assets/images/6.png',

    'assets/images/4.png',

    'assets/images/3.png',

  ];

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('Grid Examples'),

      ),

      body: GridView.builder(

        padding: const EdgeInsets.all(10.0),

        itemCount: imageUrls.length,

        gridDelegate: SliverGridDelegateWithFixedCrossAxisCount(

          crossAxisCount: 2, // 2 items per row

          crossAxisSpacing: 10.0, // Spacing between columns

          mainAxisSpacing: 10.0, // Spacing between rows

          childAspectRatio: 1.5,

        ),

        itemBuilder: (context, index) {

          return Container(

            decoration: BoxDecoration(

              border: Border.all(

                color: Colors.lime, // Border color

                width: 2.0, // Border width

              ),

              borderRadius: BorderRadius.circular(10), // Rounded corners

              boxShadow: [

                BoxShadow(

                  color: Color.fromARGB(255, 126, 148, 4).withOpacity(0.5), // Shadow color

                  spreadRadius: 2, // How much the shadow spreads

                  blurRadius: 6, // How blurry the shadow is

                ),

              ],

            ),

            child: Image.asset(

              imageUrls[index],

              fit: BoxFit.cover, // Ensures the image covers the container

            ),

          );

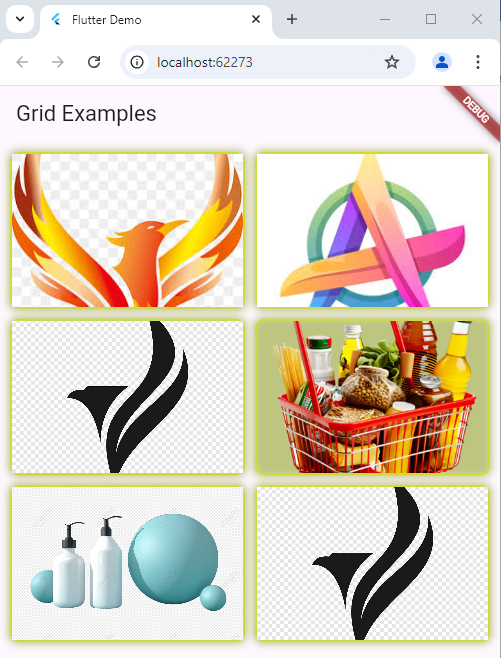
        },

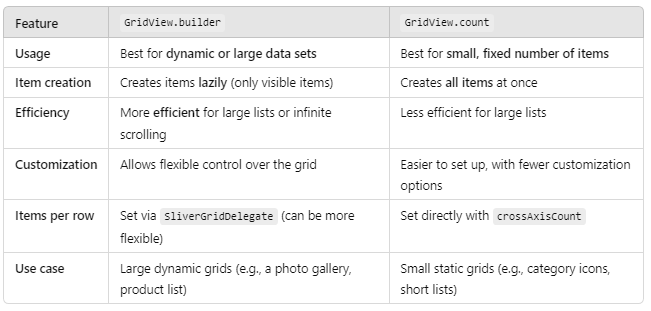
      ),

    );

  }

}

****

****

**GridView.Count**

import ‘package:flutter/material.dart’;

class gridviewCount extends StatelessWidget {

  final List<String> imageUrls = [

    ‘assets/images/2.jpg’,

    ‘assets/images/1.jpg’,

    ‘assets/images/3.png’,

    ‘assets/images/6.png’,

    ‘assets/images/4.png’,

    ‘assets/images/3.png’,

  ];

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text(‘Grid Examples’),

      ),

      body: GridView.count(

        crossAxisCount: 2, // 2 items per row

        crossAxisSpacing: 10.0, // Spacing between columns

        mainAxisSpacing: 10.0, // Spacing between rows

        padding: const EdgeInsets.all(10.0),

        childAspectRatio: 1.5, // Aspect ratio for each grid item

        children: imageUrls.map((imageUrl) {

          return Container(

            decoration: BoxDecoration(

              border: Border.all(

                color: Colors.lime, // Border color

                width: 2.0, // Border width

              ),

              borderRadius: BorderRadius.circular(10), // Rounded corners

              boxShadow: [

                BoxShadow(

                  color: Color.fromARGB(255, 126, 148, 4).withOpacity(0.5), // Shadow color

                  spreadRadius: 2, // How much the shadow spreads

                  blurRadius: 6, // How blurry the shadow is

                ),

              ],

            ),

            child: Image.asset(

              imageUrl,

              fit: BoxFit.cover, // Ensures the image covers the container

            ),

          );

        }).toList(), // Convert List<String> into List<Widget>

      ),

    );

  }

}

**GridView.count:**

Instead of GridView.builder, we now use GridView.count, which simplifies the grid creation when the number of items is known in advance.

**children:**

Instead of using itemBuilder, we directly pass the list of grid items to the children property. The imageUrls.map(...) function converts the List<String> into a list of Container widgets.

**No itemCount or itemBuilder:**

Unlike GridView.builder, GridView.count does not require an itemCount or itemBuilder. Instead, it works with a list of widgets that you supply in the children parameter.

**imageUrls** is a list of image file paths. Each path tells Flutter where to find the images in your project.

.map() is a function that goes through each item in a list and lets you decide **how to transform each item** into something else.

**(imageUrl)** is just a temporary name for each image path as the function loops through them. So:

The first time through the loop, imageUrl is 'assets/images/1.jpg'

The second time through, imageUrl is 'assets/images/2.jpg'

**GridView.extent**

GridView.extent is another way to create a grid in Flutter, and it is specifically used when you want to define a maximum width (or extent) for each item in the grid. Instead of specifying the exact number of items per row (as in GridView.count), you provide the maximum extent for each grid item, and Flutter automatically calculates how many items can fit in a row based on the screen size and available space.

**Fixed item width:** You define the maximum width (extent) for each item in the grid, and Flutter automatically arranges the items based on this value.

**Flexible number of items per row:** The number of items in each row adjusts automatically based on the screen width and the defined extent.

**Great for adaptive layouts:** It's useful when you want a flexible, responsive grid that adapts to different screen sizes without having to manually calculate the number of items per row.

**Key Properties:**

**maxCrossAxisExtent:** Specifies the maximum width of each grid item.

**crossAxisSpacing:** Space between items horizontally.

**mainAxisSpacing:** Space between items vertically.

**children:** A list of widgets that will be displayed in the grid.

**Example:**

If you specify maxCrossAxisExtent: 150 and the screen width is 300 pixels, Flutter will fit 2 items per row. If the screen width is 450 pixels, Flutter will fit 3 items per row. The grid adjusts based on the screen size.

**When to Use GridView.extent:**

* When you want to ensure that each item in your grid has a fixed maximum width, and the grid layout adjusts automatically for different screen sizes.
* For adaptive or responsive grid layouts where the number of items per row is not fixed but depends on screen size.

**Comparison with GridView.count:**

GridView.count: You manually define the number of items per row with crossAxisCount.

GridView.extent: You define the maximum width of each item, and Flutter calculates the number of items per row based on screen width.

import 'package:flutter/material.dart';

class GridviewExtent extends StatelessWidget {

  final List<String> imageUrls = [

    'assets/images/2.jpg',

    'assets/images/1.jpg',

    'assets/images/3.png',

    'assets/images/6.png',

    'assets/images/4.png',

    'assets/images/3.png',

  ];

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('Grid Examples'),

      ),

      body: GridView.extent(

        maxCrossAxisExtent: 200, // Maximum width for each item

        crossAxisSpacing: 10.0, // Spacing between columns

        mainAxisSpacing: 10.0, // Spacing between rows

        padding: const EdgeInsets.all(10.0),

        childAspectRatio: 1.5, // Aspect ratio for each grid item

        children: imageUrls.map((imageUrl) {

          return Container(

            decoration: BoxDecoration(

              border: Border.all(

                color: Colors.lime, // Border color

                width: 2.0, // Border width

              ),

              borderRadius: BorderRadius.circular(10), // Rounded corners

              boxShadow: [

                BoxShadow(

                  color: Color.fromARGB(255, 126, 148, 4).withOpacity(0.5), // Shadow color

                  spreadRadius: 2, // How much the shadow spreads

                  blurRadius: 6, // How blurry the shadow is

                ),

              ],

            ),

            child: Image.asset(

              imageUrl,

              fit: BoxFit.cover, // Ensures the image covers the container

            ),

          );

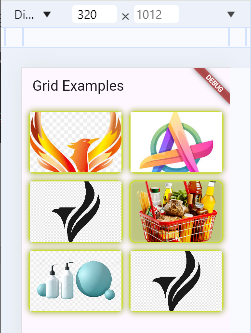
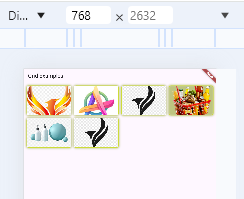
        }).toList(), // Convert List<String> into List<Widget>

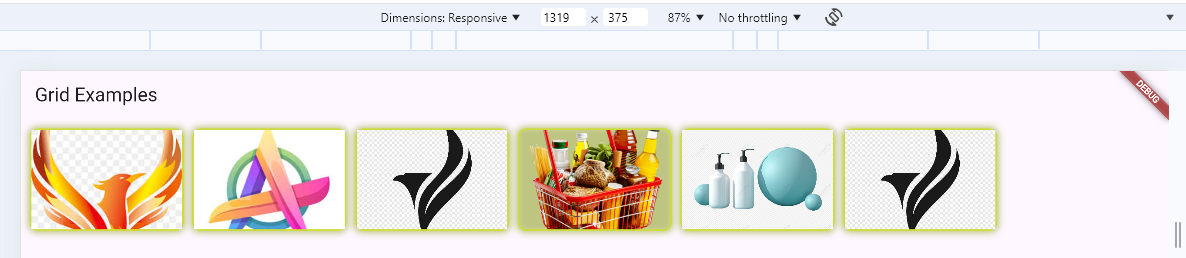
      ),

    );

  }

}

****

****

**Key Changes:**

**maxCrossAxisExtent:**

* Replaces crossAxisCount to define the maximum width for each grid item. Here, it is set to 200, meaning each grid item will have a maximum width of 200 pixels.
* The grid will automatically fit as many items as it can based on the screen size.

**Other properties:**

crossAxisSpacing, mainAxisSpacing, childAspectRatio, and children are unchanged.

**Dropdown in flutter**

The DropdownExample widget is a stateful widget because it needs to update the UI when a new value is selected.

**Dropdown Button:**

* **value:** Represents the currently selected value.
* **hint:** Displays a hint when no item is selected.
* **items:** A list of DropdownMenuItem widgets created from the items list.
* **onChanged:** A callback that gets called when the user selects a new value from the dropdown. It updates the selectedValue state.
* **isExpanded:** Makes the dropdown take up the full width of the parent widget.

**Customizing the Dropdown**

You can customize the appearance and behavior of the dropdown using various properties, such as:

* **icon:** To change the icon that appears next to the dropdown.
* **style:** To style the text of the selected item.
* **dropdownColor:** To change the background color of the dropdown menu.
* **underline:** To customize or remove the underline below the dropdown.

The StatefulWidget class is used in Flutter when you need to handle dynamic changes in the UI, like updating the state when a user interacts with widgets such as buttons, text fields, or dropdowns.

**DropdownExample extends StatefulWidget:** This defines the widget itself, which can change over time (like when the user selects a different dropdown option).

**\_DropdownExampleState:** This is the state class that holds the mutable state (in this case, the selected dropdown value) and triggers UI rebuilds using setState when the state changes.

**createState Method:**

The **createState()** method is where you instantiate the State object that corresponds to your StatefulWidget. This is required because the Flutter framework uses this method to create the State object when the widget is inserted into the widget tree.

createState is used only in stateful widgets because it creates a separate State object that can change over time.

**Think of createState like a TV remote and a TV screen:**

* The TV remote (Widget) is always the same, but the TV screen (State) changes based on the channel you select.
* createState is like setting up the connection between the remote and the screen, ensuring that pressing a button updates what you see.
* Without createState, your remote (widget) would be useless because the TV screen (state) wouldn’t update!

**State Object:**

The state object (\_DropdownExampleState in this case) contains all the logic and data related to that widget's state. This is where you handle user interactions, such as updating the dropdown value.

import 'package:flutter/material.dart';

class DropdownExample extends StatefulWidget {

  @override

  \_DropdownExampleState createState() => \_DropdownExampleState();

}

class \_DropdownExampleState extends State<DropdownExample> {

  String? selectedValue;

  final List<String> items = ['Apple', 'Banana', 'Orange', 'Grapes'];

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('Custom Dropdown Example'),

      ),

      body: Center(

        child: Padding(

          padding: const EdgeInsets.all(16.0),

          child: DropdownButton<String>(

            value: selectedValue,

            hint: Text('Select a fruit'),

            icon: Icon(

              Icons.arrow\_downward, // Custom dropdown icon

              color: Colors.blue,

            ),

            style: TextStyle(

              color: Colors.blue,  // Text style of the selected item

              fontSize: 18,

            ),

            dropdownColor: Colors.grey[200], // Background color of the dropdown

            underline: Container(  // Custom underline for the dropdown

              height: 2,

              color: Colors.blueAccent,

            ),

            items: items.map((String item) {

              return DropdownMenuItem<String>(

                value: item,

                child: Text(item),

              );

            }).toList(),

            onChanged: (String? newValue) {

              setState(() {

                selectedValue = newValue;

              });

            },

            isExpanded: true, // Expands the dropdown to take up full width

          ),

        ),

      ),

    );

  }

}

**DropdownExample extends StatefulWidget:** This widget is Stateful, meaning its state can change during its lifecycle, which is necessary for the dropdown since it will store and update the selected value.

**createState():** This method is mandatory for a StatefulWidget. It creates an instance of \_DropdownExampleState which holds the actual state of the widget.

**DropdownButton<String>:** The DropdownButton widget displays a list of options. The type <String> specifies that the dropdown will deal with string values (in this case, fruit names).

**value: selectedValue:** This is the currently selected item in the dropdown. It's linked to the selectedValue variable which gets updated when the user picks an item.

**hint: Text('Select a fruit'):** This is the placeholder text shown when no item is selected.

**icon:** Specifies the icon next to the dropdown button. We’ve customized it to use a downward arrow icon (Icons.arrow\_downward) and set the icon color to blue.

**style: TextStyle:** Customizes the text style of the selected item. In this example, the text color is blue and the font size is set to 18.

**dropdownColor:** Specifies the background color of the dropdown menu when opened. It’s set to a light gray color (Colors.grey[200]).

**underline:** This customizes the underline below the dropdown button. We use a Container to make a blue underline (Colors.blueAccent) that is 2 pixels high.

**items:**This is a property of the DropdownButton that takes a list of DropdownMenuItems, which are the options displayed in the dropdown.

**items.map((String item) { ... }):**map is a function that takes each element of the items list (which contains strings like 'Apple', 'Banana', etc.) and transforms it into something else.

In this case, it transforms each string (like 'Apple') into a DropdownMenuItem widget. map goes through every string in the list one by one.

**DropdownMenuItem<String>(...):**This is the widget that represents each item in the dropdown. It contains two main properties:

**value: item:** This is the value that the dropdown will return when the user selects this item. In our case, item is a string like 'Apple', 'Banana', etc.

**child: Text(item):** This is what will actually be displayed on the screen for the user to see. We are displaying each item as text.

**toList():**

After the map function has created a DropdownMenuItem for each string in the items list, toList() turns the resulting collection of DropdownMenuItem widgets into a list. This list is what the dropdown will use to display the options.

**Table in Flutter**

**Rows and Columns:** It organizes widgets (like Text, Icon, etc.) into rows and columns, just like a grid. You can have multiple rows, and each row can have multiple cells (columns).

**Children:** You define the content of the table by adding rows using the TableRow widget. Each TableRow contains a list of widgets (cells) that represent the columns in that row.

**Equal Width Columns:** By default, columns in the Table are distributed evenly across the available width.

**Table:** The main widget that arranges everything.

**TableBorder.all():** Adds a border around each cell.

**TableRow:** Defines a row in the table, with the content as a list of widgets (columns in that row).

**Text:** Inside the TableRow, the Text widgets represent the actual content (cells) in each column.

* All rows must have the same number of columns.
* You can customize the table by adding borders, padding, and aligning the content in different ways.

import 'package:flutter/material.dart';

class Tables extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return  Scaffold(

        appBar: AppBar(title: Text('Stylish Table Example')),

        body: Padding(

          padding: const EdgeInsets.all(16.0),

          child: Center(

            child: Table(

              border: TableBorder(

                horizontalInside: BorderSide(width: 1, color: Colors.grey),

                verticalInside: BorderSide(width: 1, color: Colors.grey),

                left: BorderSide(width: 1, color: Colors.grey),

                right: BorderSide(width: 1, color: Colors.grey),

                bottom: BorderSide(width: 1, color: Colors.grey),

              ), // Adds custom borders between rows and columns

              children: [

                // Header Row with Custom Style

                TableRow(

                  decoration: BoxDecoration(color: Colors.blueGrey.shade100),

                  children: [

                    Padding(

                      padding: const EdgeInsets.all(12.0),

                      child: Text(

                        'Item',

                        style: TextStyle(fontWeight: FontWeight.bold, fontSize: 18),

                      ),

                    ),

                    Padding(

                      padding: const EdgeInsets.all(12.0),

                      child: Text(

                        'Description',

                        style: TextStyle(fontWeight: FontWeight.bold, fontSize: 18),

                      ),

                    ),

                    Padding(

                      padding: const EdgeInsets.all(12.0),

                      child: Text(

                        'Actions',

                        style: TextStyle(fontWeight: FontWeight.bold, fontSize: 18),

                      ),

                    ),

                  ],

                ),

                // First Row

                TableRow(children: [

                  Padding(

                    padding: const EdgeInsets.all(12.0),

                    child: Row(

                      children: [

                        Icon(Icons.laptop\_mac, color: Colors.blue),

                        SizedBox(width: 10),

                        Text('Laptop'),

                      ],

                    ),

                  ),

                  Padding(

                    padding: const EdgeInsets.all(12.0),

                    child: Text('A powerful laptop with 16GB RAM.'),

                  ),

                  Padding(

                    padding: const EdgeInsets.all(12.0),

                    child: ElevatedButton(

                      onPressed: () {},

                      child: Text('Buy Now'),

                    ),

                  ),

                ]),

                // Second Row

                TableRow(children: [

                  Padding(

                    padding: const EdgeInsets.all(12.0),

                    child: Row(

                      children: [

                        Icon(Icons.phone\_android, color: Colors.green),

                        SizedBox(width: 10),

                        Text('Smartphone'),

                      ],

                    ),

                  ),

                  Padding(

                    padding: const EdgeInsets.all(12.0),

                    child: Text('Latest smartphone with AMOLED display.'),

                  ),

                  Padding(

                    padding: const EdgeInsets.all(12.0),

                    child: ElevatedButton(

                      onPressed: () {},

                      child: Text('Buy Now'),

                    ),

                  ),

                ]),

                // Third Row

                TableRow(children: [

                  Padding(

                    padding: const EdgeInsets.all(12.0),

                    child: Row(

                      children: [

                        Icon(Icons.watch, color: Colors.orange),

                        SizedBox(width: 10),

                        Text('Smartwatch'),

                      ],

                    ),

                  ),

                  Padding(

                    padding: const EdgeInsets.all(12.0),

                    child: Text('Stylish smartwatch with heart rate monitor.'),

                  ),

                  Padding(

                    padding: const EdgeInsets.all(12.0),

                    child: ElevatedButton(

                      onPressed: () {},

                      child: Text('Buy Now'),

                    ),

                  ),

                ]),

              ],

            ),

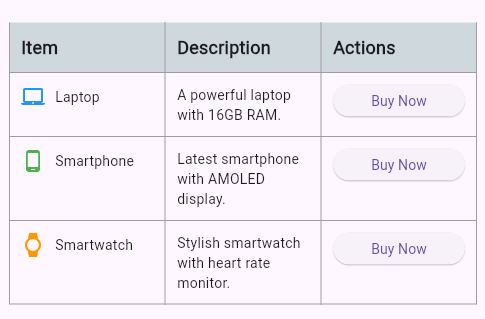
          ),

        ),

      );

  }

}

****

**POPUP Menu**

import 'package:flutter/material.dart';

class PopupMenuExample extends StatefulWidget {

  @override

  \_PopupMenuExampleState createState() => \_PopupMenuExampleState();

}

class \_PopupMenuExampleState extends State<PopupMenuExample> {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('Popup Menu Button'),

        backgroundColor: Colors.cyanAccent,

        actions: [

          PopupMenuButton<String>( // Specify the type here

            color: Colors.red,

            onSelected: (String value) {

              // Do something when an option is selected

              print(value);

            },

            itemBuilder: (BuildContext context) => [

              PopupMenuItem<String>(

                value: 'Settings', // Set the value for the item

                child: SizedBox(

                  width: 200, // Set custom width for menu

                  child: Row(

                    children: [

                      Icon(Icons.settings),

                      SizedBox(width: 10),

                      Text('Settings'),

                    ],

                  ),

                ),

              ),

              PopupMenuDivider(), // Divider between items

              PopupMenuItem<String>(

                value: 'Share', // Set the value for the item

                child: SizedBox(

                  width: 200, // Set custom width for menu

                  child: Row(

                    children: [

                      Icon(Icons.share),

                      SizedBox(width: 10),

                      Text('Share'),

                    ],

                  ),

                ),

              ),

              PopupMenuDivider(), // Divider between items

              PopupMenuItem<String>(

                value: 'Logout', // Set the value for the item

                child: SizedBox(

                  width: 200, // Set custom width for menu

                  child: Row(

                    children: [

                      Icon(Icons.logout),

                      SizedBox(width: 10),

                      Text('Logout'),

                    ],

                  ),

                ),

              ),

            ],

          ),

        ],

      ),

      body: Column(

        children: [

          SizedBox(height: 10,),

          ListTile(

            tileColor: Colors.deepOrange[50],

            leading: CircleAvatar(

              radius: 25,

              backgroundColor: Colors.blue,

              child: Text(

                'OK',

                style: TextStyle(

                  color: Colors.amber[900],

                  fontWeight: FontWeight.bold,

                ),

              ),

            ),

            title: Text('Owais Khan'),

            subtitle: Text('Subtitle for Owais'),

            trailing: PopupMenuButton<String>( // Specify the type here

              icon: Icon(

                Icons.more\_vert,

                color: Colors.amber[900],

              ),

              color: Color.fromARGB(255, 221, 234, 252),

              onSelected: (String value) {

                // Handle menu option selection

                print(value);

              },

              itemBuilder: (BuildContext context) => [

                PopupMenuItem<String>(

                  value: 'Settings', // Set the value for the item

                  child: SizedBox(

                    width: 200,

                    child: Row(

                      children: [

                        Icon(Icons.settings),

                        SizedBox(width: 20),

                        Text('Settings'),

                      ],

                    ),

                  ),

                ),

                PopupMenuDivider(), // Divider

                PopupMenuItem<String>(

                  value: 'Share', // Set the value for the item

                  child: SizedBox(

                    width: 200,

                    child: Row(

                      children: [

                        Icon(Icons.share),

                        SizedBox(width: 20),

                        Text('Share'),

                      ],

                    ),

                  ),

                ),

                PopupMenuDivider(), // Divider

                PopupMenuItem<String>(

                  value: 'Logout', // Set the value for the item

                  child: SizedBox(

                    width: 200,

                    child: Row(

                      children: [

                        Icon(Icons.logout),

                        SizedBox(width: 20),

                        Text('Logout'),

                      ],

                    ),

                  ),

                ),

              ],

            ),

          ),

        ],

      ),

    );

  }

}

**Drawer widget**

The Drawer widget in Flutter is used to create a side navigation panel that can be slid in from the side of the screen (usually from the left side, but can also be on the right). This is commonly known as a navigation drawer or side menu, which is often used in mobile apps to provide a place for secondary navigation options or extra features that don't fit in the main app bar.

Note: if you want to control height wrap that element into container then set height, like here I did with menu header.

import 'package:flutter/material.dart';

class SidePopupMenuExample extends StatefulWidget {

  @override

  \_SidePopupMenuExampleState createState() => \_SidePopupMenuExampleState();

}

class \_SidePopupMenuExampleState extends State<SidePopupMenuExample> {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('Side Popup Menu (Drawer) Example'),

      ),

      // Drawer for side popup menu

      drawer: Drawer(

        child: ListView(

          children: [

            // Drawer header

            Container(

              height: 80,

              child: DrawerHeader(

                decoration: BoxDecoration(

                  color: Colors.blue,

                ),

                child: Text(

                  'Menu Header',

                  style: TextStyle(

                    color: Colors.white,

                    fontSize: 24,

                  ),

                ),

              ),

            ),

            // Menu items

            ListTile(

              leading: Icon(Icons.settings),

              title: Text('Settings'),

              onTap: () {

                Navigator.pop(context); // Close the drawer

                // Handle your action here

              },

            ),

            Divider(),

            ListTile(

              leading: Icon(Icons.share),

              title: Text('Share'),

              onTap: () {

                Navigator.pop(context); // Close the drawer

                // Handle your action here

              },

            ),

            ListTile(

              leading: Icon(Icons.logout),

              title: Text('Logout'),

              onTap: () {

                Navigator.pop(context); // Close the drawer

                // Handle your action here

              },

            ),

          ],

        ),

      ),

      body: Center(

        child: Text('Main content area'),

      ),

    );

  }

}

**TabBar**

The TabBar widget in Flutter provides a convenient way to create tabbed navigation for your app. It is often used in combination with a TabBarView widget to display different pages or content for each tab. This approach is common in mobile apps where different sections or screens are grouped into tabs.

**Key Components of a Tab Bar in Flutter**

TabBar: This widget displays a row of tabs that users can tap to navigate between different content sections.

TabBarView: This widget is used to define the content that corresponds to each tab.

DefaultTabController: It manages the state of the TabBar and TabBarView so that they stay in sync.

**Overview of the TabBar Components**

1. TabBar

* Widget that contains the tabs (labels, icons, etc.).
* You define the tabs inside the TabBar using the tabs property, which is typically a list of Tab widgets.
* Customizable options like labelColor, indicatorColor, unselectedLabelColor, and indicatorWeight.

2. TabBarView

* Widget that controls the content for each tab.
* Each tab in the TabBar corresponds to a page or content section in the TabBarView.

3. DefaultTabController

* Manages the state of the TabBar and TabBarView.
* It automatically syncs the selected tab in TabBar with the content in TabBarView.
* Usually wraps both TabBar and TabBarView.
* If we don’t use DefaultTabController, we must:
* Create a TabController.
* Initialize it inside initState().
* Manually attach it to TabBar and TabBarView.
* Dispose of it when it's not needed.
* class HomeScreen extends StatelessWidget {
* @override
* Widget build(BuildContext context) {
* return DefaultTabController(
* length: 3, // Number of tabs
* child: Scaffold(
* appBar: AppBar(
* title: Text('TabBar in Separate Class'),
* bottom: TabBar(
* tabs: [
* Tab(icon: Icon(Icons.home), text: "Home"),
* Tab(icon: Icon(Icons.settings), text: "Settings"),
* Tab(icon: Icon(Icons.person), text: "Profile"),
* ],
* ),
* ),
* body: TabBarView(
* children: [
* HomeTab(),
* SettingsTab(),
* ProfileTab(),
* ],
* ),
* ),
* );
* }
* }
* // Separate widget class for Home Tab
* class HomeTab extends StatelessWidget {
* @override
* Widget build(BuildContext context) {
* return Center(
* child: Text("Home Tab Content"),
* );
* }
* }
* // Separate widget class for Settings Tab
* class SettingsTab extends StatelessWidget {
* @override
* Widget build(BuildContext context) {
* return Center(
* child: Text("Settings Tab Content"),
* );
* }
* }
* // Separate widget class for Profile Tab
* class ProfileTab extends StatelessWidget {
* @override
* Widget build(BuildContext context) {
* return Center(
* child: Text("Profile Tab Content"),
* );
* }
* }

**Why scaffold is under DefaultTabController**

Scaffold holds a full screen of your mobile app. When you click on different tabs it will change the screen. that changes are responsibility of DefaultTabController. That’s why

We put scaffold under this so each time user switch scaffold will be recreated.

**How Does TabBar Know Which TabBarView to Show?**

🔹 DefaultTabController automatically connects TabBar (tabs) with TabBarView (content).

🔹 It does this based on the order of tabs and views.

**Each tab is given an index automatically:**

Home = Index 0

Settings = Index 1

Profile = Index 2

**Each TabBarView child gets an index in the same order as TabBar:**

HomeTab() = Index 0 → Matches Tab(icon: Icons.home)

SettingsTab() = Index 1 → Matches Tab(icon: Icons.settings)

ProfileTab() = Index 2 → Matches Tab(icon: Icons.person)

**Floating Action Button**

Floating Action Button (FAB) can be implemented through speed dial menus or use popup menus. The most common way to provide multiple options is using a Speed Dial FAB, where the FAB expands to show additional actions. Flutter doesn’t natively include a Speed Dial FAB, but packages like flutter\_speed\_dial are often used for this purpose.

**Using Popup Menu with FloatingActionButton**

You can add a popup menu to a FAB to display multiple options when it is pressed.

class FloatingActionButtonPopUp extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('FAB with Popup Menu'),

      ),

      body: Center(

        child: Text('Press the Floating Action Button for options'),

      ),

      floatingActionButton: FloatingActionButton(

        onPressed: () {

          // You can trigger some other action here

        },

        child: PopupMenuButton<int>(

          icon: Icon(Icons.add),

          onSelected: (item) => onSelected(context, item),

          itemBuilder: (context) => [

            PopupMenuItem<int>(value: 0, child: Text('Option 1')),

            PopupMenuItem<int>(value: 1, child: Text('Option 2')),

            PopupMenuItem<int>(value: 2, child: Text('Option 3')),

          ],

        ),

      ),

    );

  }

  void onSelected(BuildContext context, int item) {

    switch (item) {

      case 0:

        print('Option 1 selected');

        break;

      case 1:

        print('Option 2 selected');

        break;

      case 2:

        print('Option 3 selected');

        break;

    }

  }

}

**Using a Speed Dial (Expanded FAB)**

Add the flutter\_speed\_dial package in your pubspec.yaml file:

dependencies:

  flutter:

    sdk: flutter

  flutter\_speed\_dial: ^6.0.0

import 'package:flutter\_speed\_dial/flutter\_speed\_dial.dart';

class FloatingActionButtonExample extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('FAB with Speed Dial'),

      ),

      body: Center(

        child: Text('Press the Floating Action Button for multiple options'),

      ),

      floatingActionButton: SpeedDial(

        animatedIcon: AnimatedIcons.menu\_close,

        backgroundColor: Colors.blue,

        overlayColor: const Color.fromARGB(255, 136, 136, 136),

        overlayOpacity: 0.5,

        children: [

          SpeedDialChild(

            child: Icon(Icons.message,color: Colors.blue,),

            label: 'Message',

            onTap: () => print('Message clicked'),

            backgroundColor: Colors.blue[100]

          ),

          SpeedDialChild(

            child: Icon(Icons.mail,color: Colors.red,),

            label: 'Mail',

            onTap: () => print('Mail clicked'),

             backgroundColor: Colors.red[100]

          ),

          SpeedDialChild(

            child: Icon(Icons.call,color: Colors.green,),

            label: 'Call',

            onTap: () => print('Call clicked'),

              backgroundColor: Colors.green[100]

          ),

        ],

      ),

    );

  }

}

You can make single FAB per screen but if you want more then you have to make it by yourself through container and additional logics

**LayoutBuilder**

In Flutter, **LayoutBuilder** is a widget that helps you build responsive layouts based on the constraints imposed by its parent. It allows you to adapt the UI dynamically depending on the available space (e.g., different screen sizes, orientations, or dynamic resizing).

LayoutBuilder provides the current constraints (such as minimum and maximum width and height) of its parent to the widget tree it builds.

The constraints describe the available space, and you can use these values to build different layouts depending on the available space

**When You Might Need Advanced Responsive Layouts**

While Flutter's built-in layouts handle many cases, you might run into issues when you need more fine-grained control. Here are some scenarios where using additional tools like LayoutBuilder, MediaQuery, or libraries for responsive design becomes important:

**Complex UIs:** As your UI becomes more complex, especially if you want different layouts for tablets vs. phones or landscape vs. portrait mode, Row and Column might not be enough. You’ll need LayoutBuilder to adjust the layout dynamically based on available space.

**Screen Size Breakpoints:** If you want a design that adapts to different screen sizes (like phones vs. tablets), you’ll need to define breakpoints, similar to how Bootstrap adjusts layout based on screen width. Row, Column, and GridView don’t natively provide breakpoints.

For example:

* A single column layout on a small screen (e.g., phone)
* A two-column layout on a larger screen (e.g., tablet)
* A three-column layout on an even bigger screen (e.g., large tablet or desktop)
* Achieving this requires more than just Row and Column and is where LayoutBuilder or a package like Responsive Framework can help.

import 'package:flutter/material.dart';

 class ResponsiveDashboard extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('Responsive Dashboard'),

      ),

      body: LayoutBuilder(

        builder: (context, constraints) {

          // Set the number of columns based on screen width

          if (constraints.maxWidth < 600) {

            return \_buildSingleColumn();

          } else if (constraints.maxWidth < 1200) {

            return \_buildTwoColumns();

          } else {

            return \_buildThreeColumnsWithSideMenu();

          }

        },

      ),

    );

  }

  // Layout for small screens (single-column layout)

  Widget \_buildSingleColumn() {

    return ListView(

      children: [

        \_buildHeader('Small Screen Layout'),

        \_buildCard('Card 1'),

        \_buildCard('Card 2'),

        \_buildCard('Card 3'),

      ],

    );

  }

  // Layout for medium screens (two-column layout)

  Widget \_buildTwoColumns() {

    return Column(

      children: [

        \_buildHeader('Medium Screen Layout'),

        Expanded(

          child: GridView.count(

            crossAxisCount: 2, // 2 columns for medium screen

            children: [

              \_buildCard('Card 1'),

              \_buildCard('Card 2'),

              \_buildCard('Card 3'),

              \_buildCard('Card 4'),

            ],

          ),

        ),

      ],

    );

  }

  // Layout for large screens (three columns + side menu)

  Widget \_buildThreeColumnsWithSideMenu() {

    return Row(

      children: [

        // Side menu for large screens

        Container(

          width: 200, // Fixed width for the side menu

          color: Colors.blueGrey,

          child: ListView(

            children: [

              \_buildMenuItem('Home'),

              \_buildMenuItem('Profile'),

              \_buildMenuItem('Settings'),

            ],

          ),

        ),

        // Main content with three columns

        Expanded(

          child: Column(

            children: [

              \_buildHeader('Large Screen Layout'),

              Expanded(

                child: GridView.count(

                  crossAxisCount: 3, // 3 columns for large screen

                  children: [

                    \_buildCard('Card 1'),

                    \_buildCard('Card 2'),

                    \_buildCard('Card 3'),

                    \_buildCard('Card 4'),

                    \_buildCard('Card 5'),

                    \_buildCard('Card 6'),

                  ],

                ),

              ),

            ],

          ),

        ),

      ],

    );

  }

  // Helper method to build a card widget

  Widget \_buildCard(String title) {

    return Card(

      margin: EdgeInsets.all(16.0),

      child: Padding(

        padding: const EdgeInsets.all(16.0),

        child: Center(

          child: Text(

            title,

            style: TextStyle(fontSize: 18),

          ),

        ),

      ),

    );

  }

  // Helper method to build a menu item in the side menu

  Widget \_buildMenuItem(String title) {

    return ListTile(

      title: Text(title),

      onTap: () {

        // Handle menu item click here

      },

    );

  }

  // Helper method to build a header for each layout

  Widget \_buildHeader(String title) {

    return Padding(

      padding: const EdgeInsets.all(16.0),

      child: Text(

        title,

        style: TextStyle(fontSize: 24, fontWeight: FontWeight.bold),

      ),

    );

  }

 }

LayoutBuilder: This widget takes the available space and provides a BoxConstraints object (constraints.maxWidth) to determine the screen width. Based on the width, different layouts are returned:

* For screens less than 600 pixels wide: a single-column layout.
* For screens between 600 and 1200 pixels: a two-column layout.
* For screens wider than 1200 pixels: a three-column layout with a side menu.

The **builder** function allows you to build a widget tree in response to certain constraints (such as screen width, height, or other conditions).

**(context, constraints)**

This is the signature of the function passed to the builder parameter. It takes two parameters: context and constraints.

**a. context**

What it represents: context is an object of type BuildContext. It represents the location of this widget in the widget tree. It carries metadata about the widget and its ancestors, and can be used to access various features (such as theme, localization, or inherited widgets).

**Usage:** You often use the context to access information about the widget tree, such as querying the size or getting the theme. It is passed automatically by Flutter.

**b. constraints**

What it represents: constraints is an object of type BoxConstraints, provided by the LayoutBuilder. It contains information about the space available for the widget (such as the maximum width and height it can use).

**Usage:** constraints allow you to determine the available size for a widget. The most common use case is to decide how to layout widgets depending on whether the screen is narrow, wide, or some other size condition.

*constraints.maxWidth and constraints.maxHeight* are commonly used to dynamically adjust the layout depending on the available space.

**Date & Time Picker**

**TextEditingController:**

**Getting the Value:** It helps you retrieve the text the user has entered into the TextField.

**Updating the Value:** You can modify the text programmatically (e.g., when the user selects a date or when you need to clear the field).

**Listening for Changes**: You can use it to listen for changes in the text (e.g., when the user types).

When the user picks a date from the date picker, we need to update the TextField with the selected date (in a formatted form, like dd/mm/yyyy). The TextEditingController allows us to set the TextField content programmatically using:

final TextEditingController \_dateController = TextEditingController();

In Flutter (and Dart), a Future is a core concept for handling asynchronous operations. It represents a value that might not be available yet but will be provided later, usually after a time-consuming operation like reading from a file, making a network request, or performing any async task.

**States of a Future:**

A Future can have one of the following states:

* Uncompleted: The task is still running, and the result is not yet available.
* Completed with a value: The task is finished, and it has returned a value.
* Completed with an error: The task has failed, and an error is returned instead of a value.

import 'package:flutter/material.dart';

class DateTimeExample extends StatefulWidget {

  @override

  \_DateTime createState() => \_DateTime();

}

class \_DateTime extends State<DateTimeExample> {

  // Variable to store the selected date

  DateTime? \_selectedDate;

  // TextEditingController to display selected date in the TextField

  final TextEditingController \_dateController = TextEditingController();

  // Function to show the date picker and update the date in the TextField

  Future<void> \_selectDate(BuildContext context) async {

    final DateTime? pickedDate = await showDatePicker(

      context: context,

      initialDate: DateTime.now(),

      firstDate: DateTime(2023),

      lastDate: DateTime(2024, 12, 31),

    );

    if (pickedDate != null) {

      setState(() {

        \_selectedDate = pickedDate;

        \_dateController.text =

            '${\_selectedDate!.day}/${\_selectedDate!.month}/${\_selectedDate!.year}'; // Format date as dd/mm/yyyy

      });

    }

  }

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('Date Picker Example'),

      ),

      body: Center(

        child: Padding(

          padding: const EdgeInsets.all(16.0),

          child: TextField(

            controller: \_dateController, // Controller to display the selected date

            decoration: InputDecoration(

              labelText: 'Select Date',

              suffixIcon: IconButton(

                icon: Icon(Icons.calendar\_today), // Calendar icon

                onPressed: () {

                  \_selectDate(context); // Open date picker on icon click

                },

              ),

              border: OutlineInputBorder(),

            ),

            readOnly: true, // Make the TextField read-only

            onTap: () {

              \_selectDate(context); // Open date picker when tapping on the TextField

            },

          ),

        ),

      ),

    );

  }

}

**Future<void>:**

* The function returns a Future, which means it's asynchronous.
* Future<void> indicates that the function performs a task that doesn't return any value (hence void), but the task might take some time to complete (such as waiting for the user to pick a date).

**Time**

import 'package:flutter/material.dart';

class TimePickerExample extends StatefulWidget {

  @override

  \_TimePickerExampleState createState() => \_TimePickerExampleState();

}

class \_TimePickerExampleState extends State<TimePickerExample> {

  // Variable to store the selected time

  TimeOfDay? \_selectedTime;

  // TextEditingController to display selected time in the TextField

  final TextEditingController \_timeController = TextEditingController();

  // Function to show the time picker and update the time in the TextField

  Future<void> \_selectTime(BuildContext context) async {

    final TimeOfDay? pickedTime = await showTimePicker(

      context: context,

      initialTime: TimeOfDay.now(), // Show current time as default

    );

    if (pickedTime != null) {

      setState(() {

        \_selectedTime = pickedTime;

        \_timeController.text =

            '${\_selectedTime!.hour}:${\_selectedTime!.minute}'; // Format time as HH:mm

      });

    }

  }

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(

        title: Text('Time Picker Example'),

      ),

      body: Center(

        child: Padding(

          padding: const EdgeInsets.all(16.0),

          child: TextField(

            controller: \_timeController, // Controller to display the selected time

            decoration: InputDecoration(

              labelText: 'Select Time',

              suffixIcon: IconButton(

                icon: Icon(Icons.access\_time), // Clock icon

                onPressed: () {

                  \_selectTime(context); // Open time picker on icon click

                },

              ),

              border: OutlineInputBorder(),

            ),

            readOnly: true, // Make the TextField read-only

            onTap: () {

              \_selectTime(context); // Open time picker when tapping on the TextField

            },

          ),

        ),

      ),

    );

  }

}

**Theme Toggling**

Theme toggling in Flutter allows you to switch between different visual themes (e.g., light and dark modes) in your app. You can define a light theme and a dark theme (or any other themes) and toggle between them using a button or some other trigger.

**ThemeData:**

The ThemeData class in Flutter is a fundamental part of theming and styling your app. It holds the configuration for the visual aspects of your app, allowing you to create a consistent look and feel across your user interface. ThemeData makes it easy to define both light and dark themes for your app, allowing users to switch between them based on their preferences.

**Common Properties of ThemeData:**

**primarySwatch:** Defines the primary color palette for the app. It is typically used for primary buttons, floating action buttons, etc.

**brightness:** Specifies whether the theme is light or dark. You can set it to Brightness.light or Brightness.dark.

**appBarTheme:** Customizes the appearance of the app bar, such as background color and text styles.

**textTheme:** Defines the text styles used throughout the app, allowing you to customize the appearance of different text elements.

**iconTheme:** Customizes the appearance of icons in the app, including color and size.

**buttonTheme:** Configures the default styles for buttons in the app.

**cardTheme:** Defines the appearance of card widgets.

Let’s create our themes using ThemeData class

**Splash Screen Widget**

import 'dart:async';

import 'package:flutter/material.dart';

import 'package:container\_widgets/SplashScreen/HomeScreen.dart';

class SplashScreen extends StatefulWidget {

  const SplashScreen({Key? key}) : super(key: key);

  @override

  \_SplashScreenState createState() => \_SplashScreenState();

}

class \_SplashScreenState extends State<SplashScreen> with SingleTickerProviderStateMixin {

  late AnimationController \_controller;

  @override

  void initState() {

    super.initState();

    // Initialize the animation controller for 360-degree rotation.

    \_controller = AnimationController(

      vsync: this,

      duration: const Duration(seconds: 3),

    )..repeat(); // Repeat the animation indefinitely.

    // Navigate to the home screen after 3 seconds.

    Timer(const Duration(seconds: 10), () {

      Navigator.pushReplacement(

        context,

        MaterialPageRoute(builder: (context) => const HomeScreen()),

      );

    });

  }

  @override

  void dispose() {

    \_controller.dispose(); // Dispose of the controller to free resources.

    super.dispose();

  }

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      backgroundColor: Colors.blue, // Background color for the splash screen.

      body: Center(

        child: Column(

          mainAxisAlignment: MainAxisAlignment.center,

          children: [

            // Rotating circular image.

            RotationTransition(

              turns: \_controller, // Use the animation controller.

              child: ClipOval(

                child: Image.asset(

                  'assets/images/1.jpg', // Path to your logo/image.

                  height: 100,

                  width: 100,

                  fit: BoxFit.cover, // Ensure the image fits within the circle.

                ),

              ),

            ),

            const SizedBox(height: 20), // Add spacing between elements.

            const Text(

              'Welcome to MyApp',

              style: TextStyle(

                color: Colors.white,

                fontSize: 24,

                fontWeight: FontWeight.bold,

              ),

            ),

          ],

        ),

      ),

    );

  }

}

**import 'dart:async';:** This allows you to use Timer, which is used for delaying actions (like navigating after 10 seconds).

Animations require a constant stream of updates to change properties (like position, size, or rotation) over time.

The **SingleTickerProviderStateMixin** acts as a provider of this ticker, ensuring smooth and efficient animations.

The **AnimationController** is the core mechanism for driving animations in Flutter.

It controls how an animation progresses over time, including starting, stopping, repeating, or reversing.

* Duration: Defines how long the animation takes to complete.
* Value: Represents the current progress of the animation (from 0.0 to 1.0).
* Methods: Provides methods like forward(), reverse(), repeat(), and stop() to control the animation.

**with SingleTickerProviderStateMixin**

**with:** This is a special keyword in Dart that adds extra features to a class. Think of it as giving your class some extra powers.

**SingleTickerProviderStateMixin:** This is the feature you're adding. It gives your class a "ticker" (like a timer) that helps with animations. It makes sure animations work smoothly and stop when they should.

The **with** keyword allows the \_SplashScreenState class to use the features provided by SingleTickerProviderStateMixin.

The SingleTickerProviderStateMixin mixin provides the **vsync** (ticker) functionality needed for animations. Without this, you would not be able to use AnimationController properly.

**mixin**

A mixin is a way to add functionality to a class without using inheritance. It allows you to reuse code across multiple classes, but unlike inheritance, it doesn't create a parent-child relationship. Instead, a mixin allows a class to "mix in" additional behavior.

In simpler terms, a mixin is like a reusable set of instructions or tools that can be added to different classes.

A **mixin** is a way to add functionality to a class without using inheritance. In other words, it allows you to reuse code across multiple classes.

**duration: const Duration(seconds: 3)**

* duration defines how long the animation should last.
* Duration(seconds: 3) means the animation will run for 3 seconds.

**vsync:** this tells the AnimationController to use the current class as the timing source for the animation.

vsync keeps the animation in sync with the screen's refresh rate, ensuring the animation updates at the right time and appears smooth.

**Example:**

If you have an animation that moves a widget across the screen, the vsync will ensure that the widget moves smoothly, updating its position at the correct times, in sync with how often the screen can refresh (its refresh rate).

**The ..repeat();** part is using cascade notation (..) to call the repeat() method on the AnimationController immediately after it is created. It tells the animation to repeat indefinitely after it finishes one cycle.

**Timer:**

Timer is a class in Dart used to schedule a function to run after a certain period of time (a delay). It helps in adding delays or timeouts in your code.

The constructor of Timer takes two arguments:

* The duration (how long to wait before running the function).
* The callback function (the function to execute after the duration).

**2. Duration(seconds: 10):**

* Duration(seconds: 10) creates a duration of 10 seconds.
* This means the timer will wait for 10 seconds before executing the function provided in the callback.

**3. Callback Function:**

The second argument is a callback function (an anonymous function or closure). In this case, it's the function that will be executed after the 10-second delay.

**Navigator.pushReplacement** is used to replace the current screen with a new one.

* It removes the current screen from the navigation stack and pushes the new screen (in this case, the HomeScreen).
* This means that after 10 seconds, the current screen (the splash screen) will be replaced by the HomeScreen.
* **MaterialPageRoute** is used to create a route to navigate to a new screen in Flutter.
* The builder property of MaterialPageRoute is a function that returns the widget to navigate to. In this case, it returns the HomeScreen.

The **dispose()** method is a lifecycle method in Flutter, and it is called when the State object of a widget is removed or destroyed.

This method is typically used to clean up resources that the widget was using, such as:

* Stopping animations.
* Closing streams.
* Disposing of controllers.

**\_controller.dispose()** is used to dispose of the AnimationController.

When you create an AnimationController, it consumes system resources. If you don't dispose of it properly, those resources are not freed, leading to memory leaks.

**RotationTransition(...):**

RotationTransition is an animated widget that rotates its child widget based on the turns value.

turns: \_controller: The turns property controls the rotation, and here it is linked to the \_controller, which is an AnimationController. This means the image will rotate continuously as the animation progresses.

**ClipOval(...):**

ClipOval is used to clip its child widget into a circular shape (an oval).

Inside the ClipOval, the child widget is an Image.asset.

The image will be displayed in a circular shape and rotate continuously (because of the animation).

--------------------------------------------------End of Widgets--------------------------------------------------------

**Basic Login System and Screen Switching**

**MyApp:** This is the entry point of your app where you define the theme.

**CustomAppBar:** This is your custom AppBar that you created.

**LoginScreen:** This is the login screen where the user can input their credentials.

**LoginButton:** A reusable button widget for the login action.

**Create a project “flutter create simple\_login”.**

**A very first step to create MyApp file for project metadata.**

import 'package:flutter/material.dart';

import 'package:simple\_login/LoginScreen.dart';

class MyApp extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return MaterialApp(

      title: 'Login System Demo',

      theme: ThemeData(

        primarySwatch: Colors.red,

      ),

      home: LoginScreen(),

    );

  }

}

* This code creates a simple Flutter app that starts with a login screen.
* The app uses a red color theme, and the LoginScreen widget is displayed as the first screen when the app runs.

**2nd step to create AppBar:**

import 'package:flutter/material.dart';

class CustomAppBar extends StatelessWidget implements PreferredSizeWidget {

  final String title;

  final List<Widget>? actions;

  final double customHeight;

  CustomAppBar({

    required this.title,

    this.actions,

    this.customHeight = 70.0,

  });

  @override

  Widget build(BuildContext context) {

    return AppBar(

      title: Text(title),

      backgroundColor: Color.fromARGB(255, 240, 250, 211),

      titleTextStyle: TextStyle(

        color: Color.fromARGB(255, 61, 1, 1),

        fontSize: 24,

      ),

      iconTheme: IconThemeData(color: Colors.white),

      centerTitle: true,

      leading: Icon(

        Icons.menu,

        color: Colors.red,

      ),

      actions: actions,

    );

  }

  @override

  Size get preferredSize => Size.fromHeight(customHeight);

}

**3rd Step Create Classes Where You Can Apply UI Tasks And Login Logic:**

Imagine you’re running a café and you need both a café layout and a manager to handle the daily operations.

**Café Layout (Café Setup):**

This is like designing the café’s interior, including tables, chairs, and decor. This represents the **LoginScreen** in our analogy—it's the visual aspect or structure that customers (users) will interact with.

**Manager (Daily Operations):**

The manager oversees the daily operations of the café, such as handling customer orders, managing staff, and ensuring everything runs smoothly. This is like the **LoginLogic** class. It handles the "logic" and "state" that make the café function properly.

**Before moving towards class here we need to understand some concept.**

* In a StatelessWidget, everything is static—once built, the UI cannot change based on user interactions or input.
* Since the login form should be able to respond to user input (such as checking the credentials and showing errors), this needs to be dynamic.
* That's why we should use a StatefulWidget for this use case, so it can handle state changes, like displaying an error message when the user enters wrong credentials or showing validation for empty fields.

Here we are making two classes LoginScreen and LoginLogic

**Purpose of LoginScreen:**

The LoginScreen class is defined as a StatefulWidget. In Flutter, StatefulWidget is generally used when the UI can change over time due to user interactions or other events. LoginScreen is meant to represent the screen or UI component where the login process happens.

**Purpose of LoginLogic:**

LoginLogic inherits from State<LoginScreen>, meaning it handles the state (logic, interactions, and UI updates) of the LoginScreen. In this case, LoginLogic is responsible for:

* Managing the state of the text fields (i.e., what the user types in the ID and password fields).
* Handling the login validation logic (checking if the username and password are correct).
* Managing error states (whether to show error messages if login fails).

LoginLogic is handling UI code here, it's actually still part of the LoginScreen widget because LoginLogic is the state for LoginScreen. When you create a StatefulWidget, the logic of how that widget behaves is kept in its State class (LoginLogic in this case).

***In Flutter, it's common practice to put both the UI and the logic in the State class for simpler cases like this one. The separation helps keep the LoginScreen class clean and focused on defining the widget, while LoginLogic handles how the UI should behave and change over time.***

import 'package:flutter/material.dart';

import 'package:simple\_login/LoginLogic.dart';

class LoginScreen extends StatefulWidget {

  const LoginScreen({super.key});

  @override

  LoginLogic createState() => LoginLogic();

}

**Concept of SUPER.KEY in Flutter**

Imagine you're sending a parcel through a delivery service. The package has a tracking number (similar to a key in Flutter), which helps the delivery system keep track of where the package is, and ensures it doesn't get mixed up with other packages.

**Example:**

* You (the sender) prepare the package (widget) and give it a unique tracking number (key).
* You hand the package to a delivery company (similar to Flutter's parent class, StatefulWidget).
* The delivery company now uses the tracking number (key) to track the package as it moves through different stages (the different widgets in Flutter).
* If the tracking number (key) is lost or not passed properly, the package may get lost, and the company won't be able to efficiently manage it.

**In our Case:**

* You are creating a widget (LoginScreen).
* The package is the widget (LoginScreen), which is part of a larger system (the Flutter widget tree).
* The tracking number is the key in Flutter, which is used to uniquely identify and manage this widget in the tree.
* The delivery company is Flutter’s system, which needs the key to efficiently manage and update widgets.

***The key ensures that Flutter knows exactly which widget it is dealing with, especially when multiple widgets of the same type are being rendered.***

***you don't need super.key if you're building a simple static widget like a login screen and don't need to uniquely identify or preserve state between multiple instances.***

***Use super.key when dealing with complex widget trees, animations, or dynamic content that might change position or number of widgets.***

**createState() role in flutter**

Imagine you are organizing a workshop, and you have a special role for a person who manages the workshop’s activities.

* You (the Organizer): You are planning the workshop (the LoginScreen).
* The Role (Manager): The manager is responsible for handling the day-to-day tasks during the workshop (the LoginLogic).

**createState()** is like appointing someone to manage a specific aspect of an event. Here, it creates an instance of LoginLogic to manage the LoginScreen.

**LoginLogic** handles the behind-the-scenes work that makes the LoginScreen functional. It is the "manager" that ensures the login functionality is operational and responds to user interactions.

import 'package:flutter/material.dart';

import 'package:simple\_login/AppBar.dart';

import 'package:simple\_login/LoginButton.dart';

import 'package:simple\_login/LoginScreen.dart';

import 'package:simple\_login/Welcome.dart'; // Import the Welcome screen

class LoginLogic extends State<LoginScreen> {

  final TextEditingController \_idController = TextEditingController();

  final TextEditingController \_passwordController = TextEditingController();

  String? idErrorMessage; // Variable to store error message for ID

  String? passwordErrorMessage; // Variable to store error message for Password

  void \_login() {

    String id = \_idController.text;

    String password = \_passwordController.text;

    bool hasError = false;

    // Check if ID field is empty

    if (id.isEmpty) {

      setState(() {

        idErrorMessage = "ID is required.";

      });

      hasError = true;

    } else {

      setState(() {

        idErrorMessage = null; // Clear error message if ID is not empty

      });

    }

    // Check if Password field is empty

    if (password.isEmpty) {

      setState(() {

        passwordErrorMessage = "Password is required.";

      });

      hasError = true;

    } else {

      setState(() {

        passwordErrorMessage = null; // Clear error message if Password is not empty

      });

    }

    // If there are errors, return early

    if (hasError) {

      return;

    }

    // Check if credentials are correct

    if (id == 'owais' && password == 'khan') {

      Navigator.push(

        context,

        MaterialPageRoute(builder: (context) => Welcome()), // Navigation to new screen

      );

    } else {

      setState(() {

        if (id != 'owais') {

          idErrorMessage = "Invalid username.";

        }

        if (password != 'khan') {

          passwordErrorMessage = "Invalid password.";

        }

      });

    }

  }

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: CustomAppBar(

        title: 'First Screen',

        actions: <Widget>[

          IconButton(

            icon: Icon(Icons.notifications, color: Colors.deepPurple),

            onPressed: () {

              print("Notification pressed");

            },

          ),

          IconButton(

            icon: Icon(Icons.settings, color: Color.fromARGB(255, 10, 10, 10)),

            onPressed: () {

              print("Settings pressed");

            },

          ),

        ],

      ),

      body: Padding(

        padding: const EdgeInsets.all(16.0),

        child: Column(

          children: [

            SizedBox(height: 80),

            Text(

              'Login Your Account',

              style: TextStyle(

                fontSize: 24,

                fontWeight: FontWeight.bold,

                color: Colors.blue,

              ),

            ),

            SizedBox(height: 80),

            TextField(

              controller: \_idController,

              decoration: InputDecoration(

                border: OutlineInputBorder(),

                labelText: 'Enter Your ID',

                errorText: idErrorMessage, // Show error message for ID

              ),

            ),

            SizedBox(height: 30),

            TextField(

              controller: \_passwordController,

              decoration: InputDecoration(

                border: OutlineInputBorder(),

                labelText: 'Enter Your Password',

                hintText: '\*\*\*\*\*\*\*\*',

                helperText: 'Your 8-digit password',

                prefixIcon: Icon(Icons.person),

                suffixIcon: Icon(Icons.check),

                errorText: passwordErrorMessage, // Show error message for Password

              ),

              obscureText: true,

            ),

            SizedBox(height: 20),

             LoginButton(onPressed: \_login), // Pass the \_login method here

          ],

        ),

      ),

    );

  }

}

extends **State<LoginScreen>** means that LoginLogic is specifically designed to manage the state for the LoginScreen widget.

**TextEditingController** manages text input in TextField widgets, with \_idController and \_passwordController specifically handling the ID and password fields, allowing for programmatic access, updates, and validation of user input.

We are using **final** because we want to fix the user input throughout the task. If we don’t use final it means we can reassign its value anywhere in our code. consider if we reassign user input in our code accidentally then we will lose the actual input user entered in textbox.

The **Navigator.push** method is used in Flutter to navigate from one screen to another. It adds a new route (or screen) onto the stack of routes managed by the Navigator. The new screen is displayed on top of the current one, and users can navigate back to the previous screen using the back button or gesture.

Now final call you Myapp class in main method.

import 'package:flutter/material.dart';

import 'package:flutter\_application\_1/MyApp.dart';

void main() {

  runApp( MyApp());

}

What if I want to add another textbox under our previous textbox? We have to make them in one stack and for stack we use column attribute.

**Column:** We use a Column widget to stack the two text fields vertically.

**SizedBox(height: 16):** This widget adds space between the two text fields to make the layout cleaner.

Two TextField widgets: Each TextField has its own label and outline, making them distinct from one another.

Now you have two text boxes on the page, and both use the same theme settings you specified earlier.

Next step is to add button under these buttons. We have two options for this just code under the textfield or just create another widget and call it on simpletextbox class.

**We are making class of our button**

import 'package:flutter/material.dart';

class LoginButton extends StatelessWidget {

  @override

  Widget build(BuildContext context) {

    return ElevatedButton(

      onPressed: () {

        // Action to perform when the button is pressed

        print("Login button pressed!");

      },

       style: ElevatedButton.styleFrom(

    backgroundColor: Colors.blue, // Updated property for background color

    foregroundColor: Colors.white, // Text color

    padding: EdgeInsets.symmetric(horizontal: 44, vertical: 12), // Button size

    shape: RoundedRectangleBorder(

      borderRadius: BorderRadius.circular(8), // Rounded corners

    ),

  ),

      child: Text('Login'),

    );

  }

}

Now call in simpletexbox calss .

  SizedBox(height: 16), // Add space before the button

            LoginButton(), // Add the login button here

--------------------------------------------------------xxxxxxxxxxxxxxxxxxxxxxxx---------------------------------------

**FireBase**

Firebase is a platform developed by Google that provides a variety of tools and services to help developers build and scale mobile and web applications. It's widely used for **real-time data management, user authentication, cloud storage, hosting**, and more. Firebase is especially popular for its ease of use, integration with Google's ecosystem, and ability to provide real-time capabilities, which are crucial for apps like chat applications, games, and collaborative tools.

**Real-time Database:**

*Firebase Realtime Database is a cloud-hosted NoSQL database* that enables developers to store and sync data in real-time between users. This is useful for chat apps, social media, and other collaborative applications where users need to see updates instantly.

**What is Real-time:**

In chat apps, when a user sends a message, it instantly appears on the other user’s screen without them needing to refresh the chat. This real-time functionality is powered by systems like Firebase.

**Example:** Imagine two friends chatting. As soon as one person sends a message, Firebase updates the database, and that message is immediately synced to the other friend’s device, creating a seamless conversation.

In apps like Uber, where users track their ride in real-time, Firebase can be used to update the driver's position on the map. As the driver moves, the location data is synced instantly with the user's device.

Example: When you request an Uber ride, you can see the driver’s real-time location on the map, and it continuously updates as the driver approaches you. This syncing ensures that both the driver and rider have an up-to-date view of each other's positions.

**NoSQL**

NoSQL stands for "**Not Only SQL,"** and it refers to a type of database that provides an alternative to traditional relational databases like MySQL, PostgreSQL, and Oracle, which use Structured Query Language (SQL) to manage and query data. NoSQL databases are designed to handle a variety of data storage needs that don't necessarily fit into the structured, table-based format used by relational databases.

**Differences Between NoSQL and SQL (Relational Databases)**

|  |  |  |
| --- | --- | --- |
| Feature | SQL (Relational Database) | NoSQL (Non-Relational Database) |
| Data Model | Tables with rows and columns (fixed schema) | Flexible models: key-value, documents, graphs, etc. |
| Schema | Fixed, predefined schema | Schema-less or dynamic schema |
| Scalability | Vertical scaling (adding resources to a single server) | Horizontal scaling (adding more servers) |
| Transaction Support | Strong ACID (Atomicity, Consistency, Isolation, Durability) | Eventual consistency, sometimes with ACID support |
| Best Use Case | Structured data, complex queries, strict consistency | Unstructured or semi-structured data, large volumes |
| Query Language | SQL (Structured Query Language) | No standard query language, may use JSON or other APIs |
| Examples | MySQL, PostgreSQL, Oracle | MongoDB, Cassandra, DynamoDB, Redis |

**Main services Firebase offers:**

1. **Databases**
2. **Authentication**
3. Cloud Storage
4. Hosting
5. Cloud Functions

**Firestore**

Firestore is a cloud-based NoSQL database provided by Firebase that you can use in your Flutter apps. It allows you to store, sync, and query data in real time. Firestore organizes data in collections and documents, where:

* Collections are like folders.
* Documents store data in key-value pairs, similar to a JSON object.

Firestore is great for:

* Real-time updates (data syncs instantly across devices).
* Offline support (data can be accessed even when offline).

Firestore's offline support means that your Flutter app can still function and access data even when the device is not connected to the internet.

**Read operations:** Data that was previously fetched while online is cached and can be accessed when offline.

**Write operations:** You can add or update documents while offline. These changes will be synced to the server when the connection is restored.

**Example:**

In a chat app, users can send and receive messages even if they temporarily lose internet connectivity:

**Online:** Messages are instantly synced between users in real-time.

**Offline:** Users can still type and send messages, which are stored locally on their device. When the connection is restored, Firestore syncs these messages with the server, and other users receive them.

**Google Analytics:**

Google Analytics helps you understand how users engage with your Flutter app by collecting data like:

**User Behavior:** Track how users navigate your app, which screens they visit, how long they stay, and where they drop off.

**Events:** Record specific actions users take, such as button clicks, form submissions, or purchases.

**Audience Insights:** Learn about your users' demographics, such as age, gender, location, and interests.

**Crash Reporting:** Automatically track app crashes and stability issues, helping you identify and fix problems.

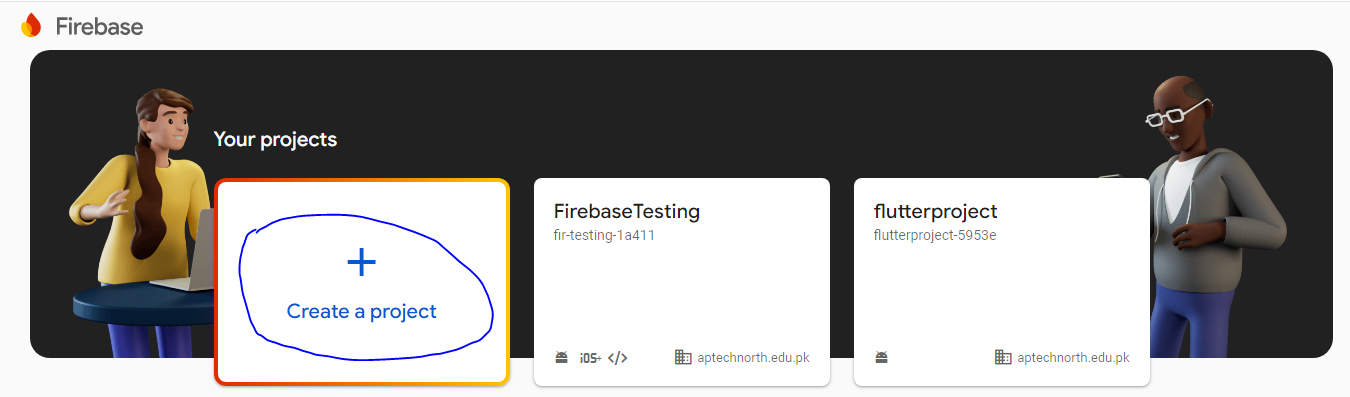
**User Retention:** See how often users return to your app and how long they engage over time.

**Conversions:** Track important user actions like completing a signup, making a purchase, or any custom-defined action that matters to your app.

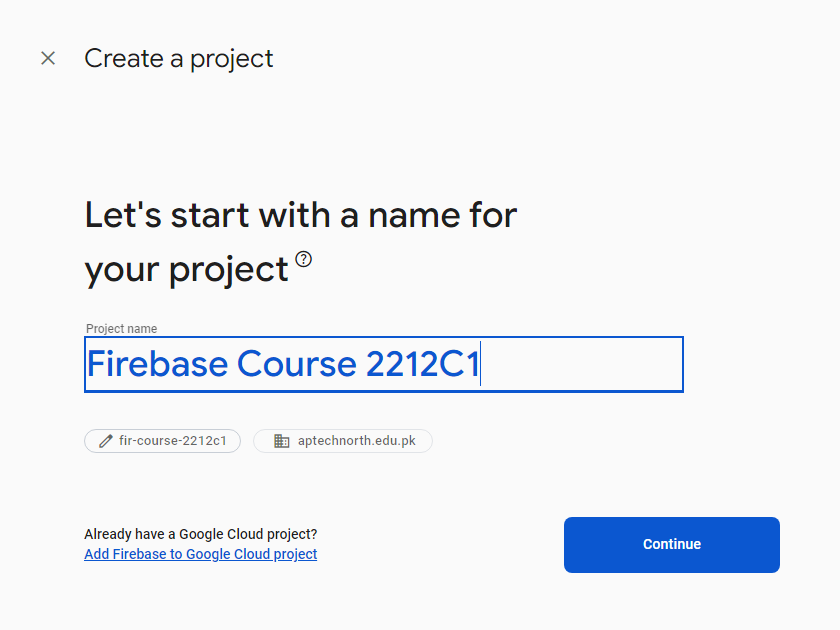
Firebase Setup

**Firebase account:** You need a Google account to access the Firebase console.

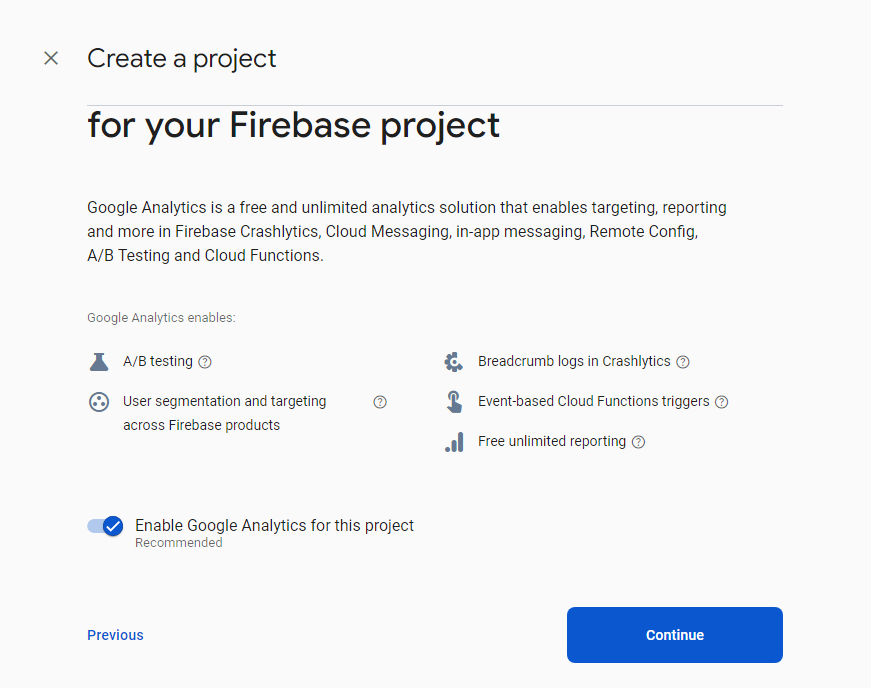
1. Go to <https://console.firebase.google.com/>
2. Click on create a project



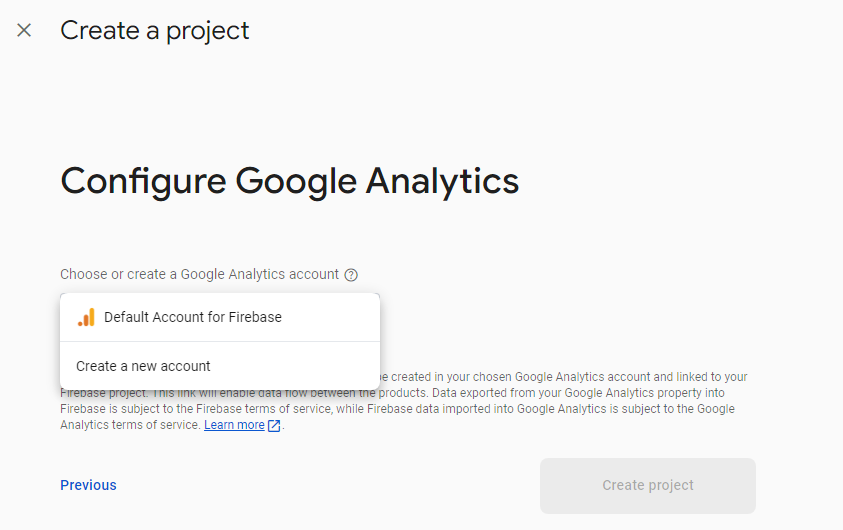
1. Enter your Project Name and click on continue:



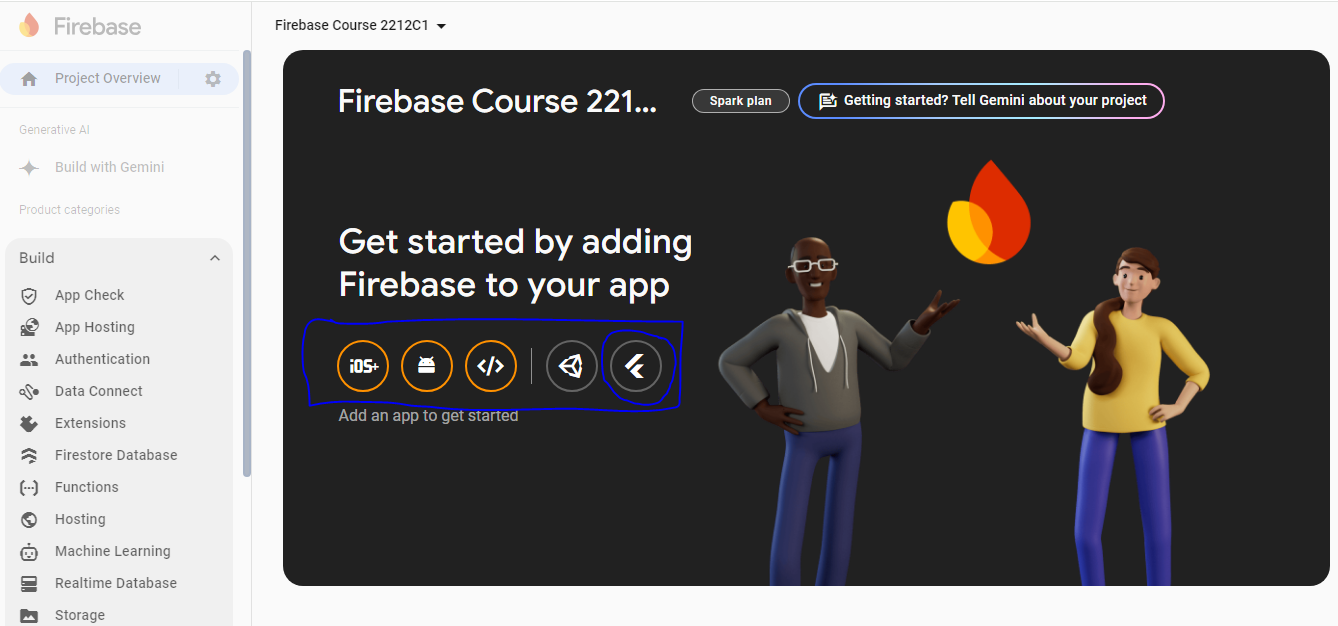
1. Enable or disable google analytics of your project and then click on continue.



1. Select account for getting analytics. here I choose my default firebase account from which I am logged in.click on create project.

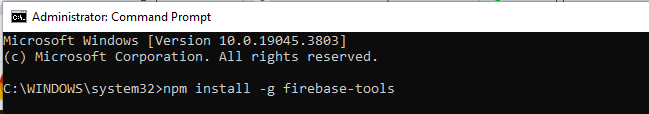


1. Project has been created now it is showing created project screen. Now its time to select option that for what we have created this project here are multiple options like android, iOS, web, flutter we are going to click on flutter.



1. now it is showing which pre requisite you are required. keep in mind GIT should be in computer. Next thing we need is flutter CLI. you can download it through various options but if you have node in your computer then NPM is best option. Open cmd and run it as administrator and enter the following command to install firebase cli globally.

npm install -g firebase-tools

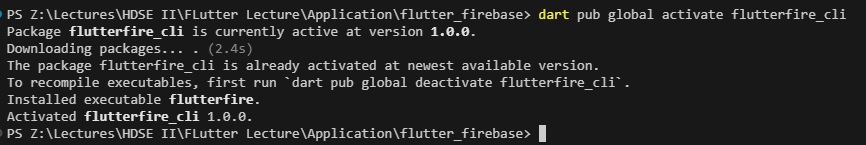


1. Once it get installed set the environment variable to use that cli in any project .

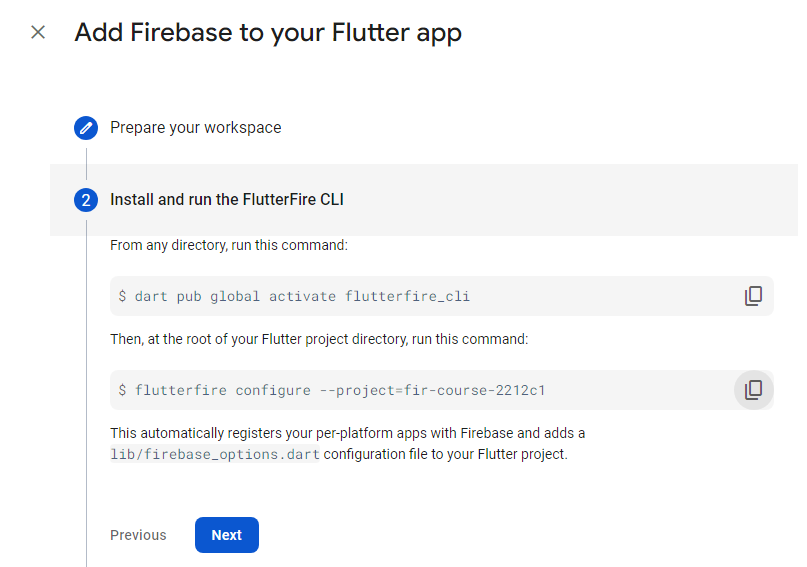
**C:\Users\owais\AppData\Local\Pub\Cache\bin**

1. Now open your flutter project in your IDE. (In my case I am using VS Code).open terminal and run following command.

**dart pub global activate flutterfire\_cli**

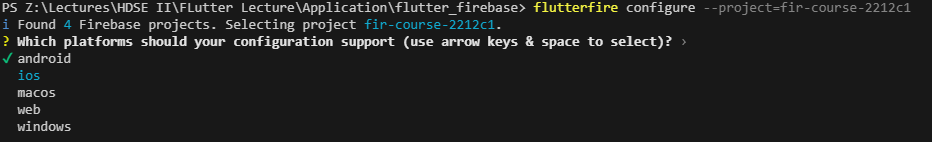
****

1. Once above command run successfully. Run another command given below. Keep in mind these both commands will be generated by firebase and will be different for each project like following:

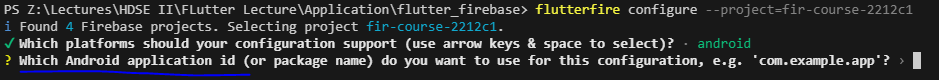


**flutterfire configure --project=fir-course-2212c1**

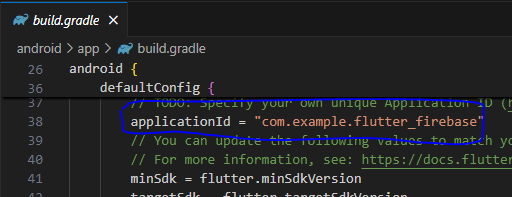
when you run this command, it will ask that for what OS you are going to make app. We will select android. use space bar for selecting and deselecting.



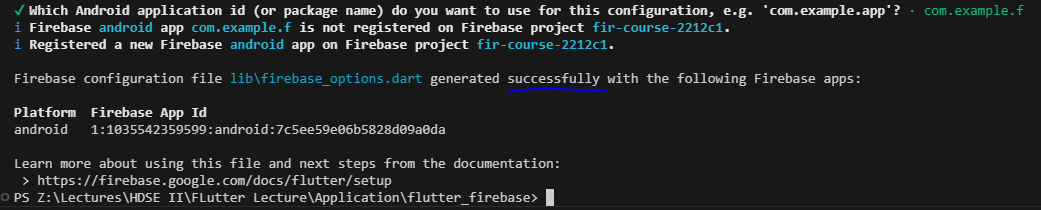
Next it will ask about application id of your project .like:



To answer this question go to **android/app/build.gradle** file. And copy application id from there and paste it in terminal.



Now flutter project has been successfully integrated with our firebase project.



1. Now you can check in lib folder, firebase has added a file firebase\_options. dart.but it will have error “Target of URI doesn't exist: 'package:firebase\_core/firebase\_core.dart'.” this error is coming under import of firebase core . we have to add firebase core in our this file. For this go to <https://pub.dev/packages/firebase_core>. And copy name of file **firebase\_core: ^3.6.0.**then paste this name in your pubsec.yaml file under dependencies(sometimes version issues of firebase core and firestore so use suggested versions).

**dependencies:**

**flutter:**

**sdk: flutter**

**firebase\_core: 2.32.0**

**cloud\_firestore: ^4.11.0**

**after changes in file run flutter pub get command to embed changes.**

1. Now initialize firebase in your main.dart file. Open firbase\_options file and copy following lines :

apiKey: 'AIzaSyAVcWBwGZ7\_\_OxYvjiatGpbl6nRrzse7Js',

    appId: '1:1035542359599:android:7c5ee59e06b5828d09a0da',

    messagingSenderId: '1035542359599',

    projectId: 'fir-course-2212c1',

now paste these lines under main method after making it async

void main() async {

    WidgetsFlutterBinding.ensureInitialized();

  // Check if Firebase is already initialized

  if (Firebase.apps.isEmpty) {

    await Firebase.initializeApp(

      options: const FirebaseOptions(

        apiKey: 'AIzaSyA41jp38NNN7ntMjOPuKCsbuQhdhn0uMvc',

        appId: '1:862025902394:android:2868875f8504a507d04803',

        messagingSenderId: '862025902394',

        projectId: 'fir-testing-1a411',

      ),

    );

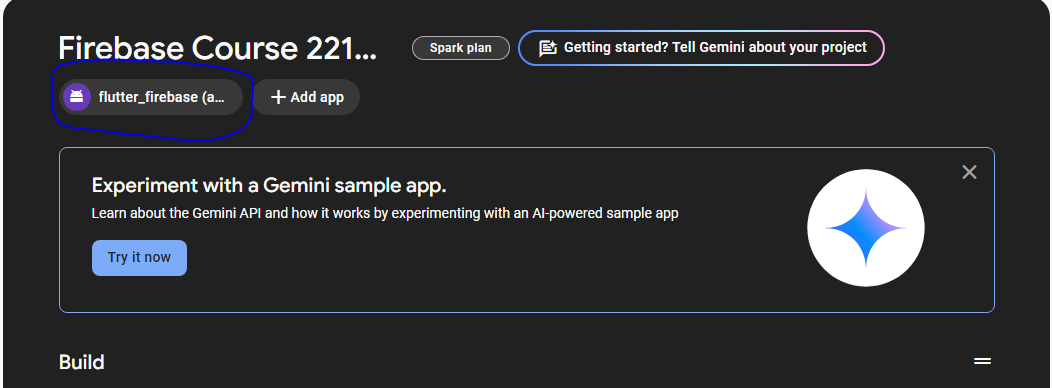
  }

  runApp(const MyApp());

}

Now run your flutter app it should run properly with his default code (increment code).

At this point flutter project has been synchronized with firebase .



**------------------------------------------------------XXXXXX---------------------------------------------------------------**

**Crud Operation in Flutter**

1. Create database on firebase and the configured your project with firestore as discussed above.
2. We need 5 textboxes to get input from user. To get that input from text field we need controllers who can listen changes in textboxes. So for each textbox I am adding controllers.

final TextEditingController \_firstNameController = TextEditingController();

  final TextEditingController \_lastNameController = TextEditingController();

  final TextEditingController \_emailController = TextEditingController();

  final TextEditingController \_ageController = TextEditingController();

  final TextEditingController \_addressController = TextEditingController();

The **final** keyword ensures that once the TextEditingController is initialized, you cannot reassign it to another controller. This is important because the TextEditingController should be consistent throughout the lifecycle of the widget to correctly manage the text input.

1. Now I need to make a connection to firebase to add, update, delete and select data. Her I have to tell in my code which collection’s (like table in sql) data I want to manipulate.here we come with concept of **CollectionReference** class**. I made user name object of this class.**

**FirebaseFirestore**

FirebaseFirestore is like a database that stores data in the cloud.it's where your app’s data is saved and retrieved from.

**instance**

instance means you're creating a connection to the Firestore database.It's like saying, "Hey, I want to start using the Firestore database right now."

**collection('users')**

collection refers to a folder in the database.nside this folder, you can store many documents (like files).

'users' is the name of the collection (or folder) where you'll save or get data.

For example, this folder might store details about your app's users (names, emails, etc.).

final CollectionReference users = FirebaseFirestore.instance.collection('users');

***here connection and input handling is done now come towards designing phase .***

body: Padding(

        padding: const EdgeInsets.all(16.0),

        child: Column(

          crossAxisAlignment: CrossAxisAlignment.start,

          children: [

            // Text fields for each attribute

            TextField(

              controller: \_firstNameController,

              decoration: InputDecoration(labelText: 'First Name'),

            ),

            TextField(

              controller: \_lastNameController,

              decoration: InputDecoration(labelText: 'Last Name'),

            ),

            TextField(

              controller: \_emailController,

              decoration: InputDecoration(labelText: 'Email'),

            ),

            TextField(

              controller: \_ageController,

              decoration: InputDecoration(labelText: 'Age'),

              keyboardType: TextInputType.number, // Allow only numbers

            ),

            TextField(

              controller: \_addressController,

              decoration: InputDecoration(labelText: 'Address'),

            ),

5 text fields have been added and their controllers are also assigned so each controller can listen each textboxes changes.

Next we need a button to insert data to our firebase database .after giving gap we are adding elevated button and on press method will run our custom created method **\_create**

SizedBox(height: 20),

            ElevatedButton(

              onPressed: \_create,

              child: Text('Create'),

            ),

Let’s create a \_create method.

void \_create() async {

    try {

      await users.add({

        'first\_name': \_firstNameController.text,

        'last\_name': \_lastNameController.text,

        'email': \_emailController.text,

        'age': int.parse(\_ageController.text), // Convert age to an integer

        'address': \_addressController.text,

      });

      // Clear the text fields after creation

      \_firstNameController.clear();

      \_lastNameController.clear();

      \_emailController.clear();

      \_ageController.clear();

      \_addressController.clear();

    } catch (e) {

      print(e);

    }

  }

In first part of code, we are adding text fields data to users’ collection. And in the end, we are clearing data from text controllers.

Next part is when user add on create method that data should save in firestore as well as show under create button in a table.

I am using expanded under create button because :

Expanded widget is used to ensure that the widget inside it takes up all the available space in the parent Column after the other widgets have been laid out. Specifically, the Expanded widget is wrapping the \_buildUserTable() widget. The Expanded widget makes the \_buildUserTable() (which returns a StreamBuilder containing the DataTable) expand to fill the remaining available space within the Column layout.

 SizedBox(height: 20),

            Expanded(child: \_buildUserTable()),

Next task is to create \_buildUserTable where all data will be fetched from firestore and decorated in a table. We need:

**StreamBuilder:**

This listens to a stream of data (in this case, a Firebase Firestore users’ collection). It continuously updates the UI whenever the data changes (like when new users are added, updated, or deleted).

the data being listened to is a QuerySnapshot from Firestore, which holds all the user documents (data) from the users collection in Firestore.

**QuerySnapshot(holding actual data from firestore)**

A QuerySnapshot represents a collection of documents (data) retrieved from a query in Firebase Firestore.it refers to the result of fetching all the user records from the Firestore users’ collection. The snapshot contains the data for all users (documents) that match the query.

When you access snapshot.data!.docs, you're looking at a list of user documents from Firestore.

**snapshots ():**

snapshots () is a Firestore function that returns a stream of data. This stream sends updates whenever there’s a change in the Firestore collection (such as adding, updating, or deleting documents).

users.snapshots() to get real-time data updates from the Firestore **users** collection. It gives you a live feed of changes from the Firestore database, which the StreamBuilder listens to and uses to update the UI.

*snapshots () continuously stream updates, StreamBuilder listens for those updates and rebuilds the UI with the latest QuerySnapshot data from Firestore.*

**Loading Indicator:**

If the data isn’t ready yet (for example, when it’s still being fetched from Firestore), it shows a loading spinner (CircularProgressIndicator) in the center of the screen.

**Scrollable Table:**

Once the data is available, it creates a horizontally scrollable table (SingleChildScrollView with scrollDirection: Axis.horizontal). This ensures the table can be scrolled if it becomes too wide for the screen.

**DataTable:**

This widget displays data in a table format. It has predefined column headers:

First Name, Last Name, Email, Age, Address, and Actions.

**Rows:**

Each row of the table represents a user, and it pulls information like first\_name, last\_name, email, age, and address from Firestore documents.If any of the data is missing, it shows 'N/A' (Not Available) instead.

**snapshot.data!.docs:** This retrieves all the documents (user records) from the Firestore query. Each document is a record from the users collection.

**.map((doc):** The map() function loops over every document (doc) and applies a function to each one. In this case, it's going to create a DataRow for each document.

Firestore returns a list of users as documents (let's say 3 users). Each document has data like the user's name, email, and age. The map() function will go through each user document one by one, creating a row in the table for each user.

So, for each user (doc), you will get a row in the table with that user's data, and all rows will be listed in the DataTable.

  Widget \_buildUserTable() {

  return StreamBuilder<QuerySnapshot>(

    stream: users.snapshots(),

    builder: (context, snapshot) {

      if (!snapshot.hasData) {

        return Center(child: CircularProgressIndicator());

      }

      return SingleChildScrollView(

        scrollDirection: Axis.horizontal, // Horizontal scroll if table is too wide

        child: SingleChildScrollView(

          scrollDirection: Axis.vertical, // Vertical scroll if table is too tall

          child: DataTable(

            columns: [

              DataColumn(label: Text('First Name')),

              DataColumn(label: Text('Last Name')),

              DataColumn(label: Text('Email')),

              DataColumn(label: Text('Age')),

              DataColumn(label: Text('Address')),

              DataColumn(label: Text('Actions')),

            ],

            rows: snapshot.data!.docs.map((doc) {

              return DataRow(

                cells: [

                  DataCell(Text(doc['first\_name'] ?? 'N/A')),

                  DataCell(Text(doc['last\_name'] ?? 'N/A')),

                  DataCell(Text(doc['email'] ?? 'N/A')),

                  DataCell(Text(doc['age']?.toString() ?? 'N/A')),

                  DataCell(Text(doc['address'] ?? 'N/A')),

                  DataCell(Row(

                    mainAxisAlignment: MainAxisAlignment.spaceEvenly,

                    children: [

                      IconButton(

                        icon: Icon(Icons.edit, size: 18),

                        onPressed: () => \_update(doc),

                      ),

                      IconButton(

                        icon: Icon(Icons.delete, size: 18),

                        onPressed: () => \_delete(doc.id),

                      ),

                    ],

                  )),

                ],

              );

            }).toList(),

          ),

        ),

      );

    },

  );

}

**Update Method:**

The \_update function here is designed to update a user’s information in Firebase Firestore. It first displays a dialog box with input fields for updating information and then saves the updated data back to Firestore when the "Update" button is clicked.

* To update data we need a specific date which needs to be updated not a whole collection.so for getting that data we need something which can give us that data from firestore.
* Here we come with **DocumentSnapshot.** DocumentSnapshot is a temporary view of a single document's data in Firestore, which you can use to read its contents.
* You can use DocumentSnapshot to get or display data, but you can’t modify the original document through it directly (you’d need to use update for that).
* When displaying your list of users, you’ll likely show each one as a row or list item.
* To update a specific document, you typically select or click on a particular item (like a row with user data).
* When you click on this item, your app captures that specific document’s DocumentSnapshot (including its unique ID).
* When you tap or click on a specific user in the list, like “John Doe,” the app captures the DocumentSnapshot of that selected user.
* Each document in Firestore has a unique ID, so when you tap “John Doe,” the app knows it’s selecting only John Doe’s document based on his unique ID.
* After selecting the user, the app passes that specific DocumentSnapshot (containing the unique ID and all fields of John Doe) to the \_update function.
* Now, the \_update function knows it has only John Doe’s data to work with, even if there are thousands of other documents in Firestore.

void \_update(DocumentSnapshot doc) async {

    \_firstNameController.text = doc['first\_name'];

    \_lastNameController.text = doc['last\_name'];

    \_emailController.text = doc['email'];

    \_ageController.text = doc['age'].toString();

    \_addressController.text = doc['address'];

Now I want to open a popup with all these data.

**await showDialog(...):** This line creates and shows the pop-up box (dialog) on top of the screen.

**context: context:** This tells the dialog to open in the part of the app where you clicked or tapped, so it’s connected to the current screen.

**AlertDialog:** This is a special type of widget in Flutter that’s ready-made for dialogs. It helps us easily create a box with a title, content, and buttons.

Imagine a pop-up window that has a title (at the top), content (in the middle), and buttons (at the bottom). This is what AlertDialog does for us.

**content: Column(...):** This part is where we put the main content of the dialog—in this case, input fields for user information.

**Column:** Flutter’s Column widget stacks things vertically, like piling objects one on top of another.

**mainAxisSize: MainAxisSize.min:** This part makes sure that the dialog box only takes up enough space to fit the fields and does not stretch across the whole screen.

Think of Column like arranging multiple fields (e.g., name, email, age) in a single column, so they show up neatly, one after the other, without wasting screen space.

await showDialog(

      context: context,

      builder: (context) => AlertDialog(

        title: Text("Update User"),

        content: Column(

          mainAxisSize: MainAxisSize.min,

          children: [

            TextField(

              controller: \_firstNameController,

              decoration: InputDecoration(labelText: 'First Name'),

            ),

            TextField(

              controller: \_lastNameController,

              decoration: InputDecoration(labelText: 'Last Name'),

            ),

            TextField(

              controller: \_emailController,

              decoration: InputDecoration(labelText: 'Email'),

            ),

            TextField(

              controller: \_ageController,

              decoration: InputDecoration(labelText: 'Age'),

              keyboardType: TextInputType.number,

            ),

            TextField(

              controller: \_addressController,

              decoration: InputDecoration(labelText: 'Address'),

            ),

Now the final thing is to update firestore with user changes:

The actions part is where we add buttons to the bottom of the dialog box (pop-up). Here, we have one button labeled “Update.”

**Updating the Data in Firebase:**

await users.doc(doc.id).update({...}): This line goes to Firestore and updates the user’s data.

doc.id: Each user’s data is stored with a unique ID. Using this ID, we make sure we update only this specific user’s document.

Inside update({...}): Each field (first\_name, last\_name, etc.) is updated with the new text the user has entered in the dialog box.

 actions: [

          ElevatedButton(

            onPressed: () async {

              await users.doc(doc.id).update({

                'first\_name': \_firstNameController.text,

                'last\_name': \_lastNameController.text,

                'email': \_emailController.text,

                'age': int.parse(\_ageController.text),

                'address': \_addressController.text,

              });

              \_firstNameController.clear();

              \_lastNameController.clear();

              \_emailController.clear();

              \_ageController.clear();

              \_addressController.clear();

              Navigator.of(context).pop();

            },

            child: Text('Update'),

          )

        ],

      ),

    );

  }

**Delete User:**

Let’s say you have a delete button next to each user’s data in the list. When you press the delete button, it triggers a function that calls \_delete and passes the specific ID of the selected user.

When you click the delete button for a specific user, the app passes that user's unique ID into the \_delete function so it knows which user to delete.

  void \_delete(String id) async {

    await users.doc(id).delete();

  }

**Shared Preferences:**

In Flutter, **Shared Preferences** is a way to store small amounts of data locally on a device in the form of key-value pairs. It's commonly used for saving simple, app-wide settings or user preferences that the app can remember even after it's closed.

**Real-World Examples of Shared Preferences:**

**User Login Status:**

To check if a user has already logged in, you can store a boolean value (like isLoggedIn = true) when they first log in. Each time the app opens, it can check this value and skip the login screen if it’s true.

**Language Selection:**

For multilingual apps, you can store the selected language code (like “en” for English or “fr” for French) and load it when the app starts to display content in the chosen language.

**User Preferences in Forms:**

If your app has forms where users regularly select the same options, you can store these options to pre-fill fields or set defaults for convenience.

**Add Shared Preferences to your project: In pubspec.yaml, add:**

dependencies:

shared\_preferences: ^2.0.6

**Authentication System in Flutter**

**Requirements:**

1. **Firebase Project Setup**
2. **Dependencies in pubspec.yaml**

dependencies:

firebase\_core: latest\_version # Required to initialize Firebase

firebase\_auth: latest\_version # Required for Firebase authentication

1. **Authentication Methods**

Firebase Authentication supports multiple login methods. For a basic setup, you’ll likely use:

* Email and Password Authentication
* Optionally, Google Sign-In or Facebook Login (requires additional configuration).

1. **Shared Preferences**

If your application requires users to log in every time without an option to stay logged in, then Shared Preferences may not be necessary.

**AuthProvider**

The AuthProvider class provides centralized authentication logic, session persistence, and state management for the app. It handles all interactions with Firebase Authentication, manages persistent login preferences via SharedPreferences, and updates the UI reactively by notifying listeners of any state changes. This structure makes it easier to maintain authentication logic in one place, keeping the app's authentication state and preferences synchronized across sessions and simplifying the UI’s ability to respond to changes in authentication status.

// AuthProvider.dart

// Add this for web-specific storage

import 'package:firebase\_auth/firebase\_auth.dart';

import 'package:flutter/material.dart';

import 'package:shared\_preferences/shared\_preferences.dart';

class AuthProvider extends ChangeNotifier {

  final FirebaseAuth \_auth = FirebaseAuth.instance;

  bool \_isLoggedIn = false;

  bool \_rememberMe = false;

  bool get isLoggedIn => \_isLoggedIn;

  bool get rememberMe => \_rememberMe;

  AuthProvider() {

    initializeAuth();

  }

  Future<void> initializeAuth() async {

    SharedPreferences prefs = await SharedPreferences.getInstance();

    \_rememberMe = prefs.getBool('rememberMe') ?? false;

    try {

      if (\_rememberMe) {

        await \_auth.setPersistence(Persistence.LOCAL);

      } else {

        await \_auth.setPersistence(Persistence.SESSION);

      }

    } catch (e) {

      print("Error setting persistence: $e");

    }

    \_isLoggedIn = \_rememberMe && \_auth.currentUser != null;

    notifyListeners();

  }

 Future<String?> login(String email, String password, bool rememberMe) async {

  try {

    // Clear any existing session before attempting login

    await \_auth.signOut();

    // Set persistence based on "Remember Me"

    await \_auth.setPersistence(rememberMe ? Persistence.LOCAL : Persistence.SESSION);

    // Attempt sign-in

    await \_auth.signInWithEmailAndPassword(email: email, password: password);

    // Update local state

    \_isLoggedIn = true;

    \_rememberMe = rememberMe;

    // Store "Remember Me" preference

    SharedPreferences prefs = await SharedPreferences.getInstance();

    prefs.setBool('rememberMe', \_rememberMe);

    // Notify listeners of change

    notifyListeners();

    return null; // Login successful

  } catch (e) {

    print("Login error: ${e.toString()}"); // Debug line

    // Return a single, generic error message for any login failure

    return 'invalid\_credentials';

  }

}

  void logout() async {

    await \_auth.signOut();

    \_isLoggedIn = false;

    SharedPreferences prefs = await SharedPreferences.getInstance();

    prefs.setBool('rememberMe', false);

    print("Remember Me Cleared on Logout");

    notifyListeners();

  }

  Future<void> signup(String email, String password) async {

    try {

      await \_auth.createUserWithEmailAndPassword(

          email: email, password: password);

      \_isLoggedIn = true;

      notifyListeners();

    } catch (e) {

      throw "Signup failed: $e";

    }

  }

}

**AuthProvider that extends ChangeNotifier**.

This class will manage authentication logic and notify listeners (like UI components) when there's a change (like logging in or out).

final FirebaseAuth \_auth = FirebaseAuth.instance;

This line creates an instance of FirebaseAuth.It provides access to Firebase authentication methods (like signing in and out).

bool \_isLoggedIn = false;

bool \_rememberMe = false;

These are private variables that keep track of the user's login status and whether to remember the user.

\_isLoggedIn: A flag that indicates if the user is currently logged in.

\_rememberMe: A flag that indicates if the user wants to be remembered for future logins.

bool get isLoggedIn => \_isLoggedIn;

bool get rememberMe => \_rememberMe;

These are public methods that allow other parts of the code to access the \_isLoggedIn and \_rememberMe variables.This makes it easy to check the login status and remember setting from other parts of the app.

**initializeAuth Method:** This method initializes the authentication state when the app starts.It checks if the user chose to be remembered and sets the appropriate authentication persistence.

The method checks if the user has previously opted to be remembered. If they have, it can automatically log them in or set the appropriate persistence setting for the session. This enhances the user experience by allowing users to stay logged in across sessions if they chose that option.

Future<void> initializeAuth() async {

    SharedPreferences prefs = await SharedPreferences.getInstance();

    \_rememberMe = prefs.getBool('rememberMe') ?? false;

    try {

      if (\_rememberMe) {

        await \_auth.setPersistence(Persistence.LOCAL);

      } else {

        await \_auth.setPersistence(Persistence.SESSION);

      }

    } catch (e) {

      print("Error setting persistence: $e");

    }

    \_isLoggedIn = \_rememberMe && \_auth.currentUser != null;

    notifyListeners();

  }

SharedPreferences prefs = await SharedPreferences.getInstance();

This line initializes an instance of SharedPreferences, which is a way to store simple data persistently in the application.

\_rememberMe = prefs.getBool('rememberMe') ?? false;

This line retrieves the value associated with the key 'rememberMe' from shared preferences and assigns it to the \_rememberMe variable.

prefs.getBool('rememberMe') tries to get a boolean value stored under the key 'rememberMe'.

If there is no value found (i.e., the user has never set this preference), it defaults to false due to the ?? operator. This means if prefs.getBool('rememberMe') is null, \_rememberMe will be set to false, indicating that the user does not want to be remembered.

\_isLoggedIn = \_rememberMe && \_auth.currentUser != null;

This line determines if the user is currently logged in based on their preferences.

It sets \_isLoggedIn to true if both conditions are met:

* \_rememberMe is true (the user chose to be remembered), and
* \_auth.currentUser != null (there is a user currently signed in).
* If either condition is false, \_isLoggedIn will be false.

Since AuthProvider extends **ChangeNotifier**, calling notifyListeners() will update any UI components that are listening to changes in this provider, allowing the UI to reactively display the current authentication state.

**Login Method:**

Future<String?> login(String email, String password, bool rememberMe) async {

    try {

      // Clear any existing session before attempting login

      await \_auth.signOut();

      // Set persistence based on "Remember Me"

      await \_auth

          .setPersistence(rememberMe ? Persistence.LOCAL : Persistence.SESSION);

      // Attempt sign-in

      await \_auth.signInWithEmailAndPassword(email: email, password: password);

      // Update local state

      \_isLoggedIn = true;

      \_rememberMe = rememberMe;

      // Store "Remember Me" preference

      SharedPreferences prefs = await SharedPreferences.getInstance();

      prefs.setBool('rememberMe', \_rememberMe);

      // Notify listeners of change

      notifyListeners();

      return null; // Login successful

    } catch (e) {

      // Return a single, generic error message for any login failure

      return 'invalid\_credentials';

    }

  }

The **login function** is responsible for handling user login in the application. It takes an email, password, and a "Remember Me" option as inputs. The function performs the following tasks:

* It logs out any existing user session to ensure a clean login attempt.
* It sets the session persistence based on whether the user wants to be remembered.
* It tries to sign the user in using the provided email and password.
* If the sign-in is successful, it updates the user's login state, saves their preference for "Remember Me," and notifies any parts of the app that might need to react to this change.
* If there's an error (like invalid credentials), it returns a generic error message.

**await \_auth.signOut();**

This line logs out the currently authenticated user to clear any existing sessions before attempting a new login.

**await \_auth.setPersistence(rememberMe ? Persistence.LOCAL : Persistence.SESSION);**

* This line sets the persistence of the authentication state based on whether the user wants to be remembered.
* If rememberMe is true, it sets persistence to LOCAL, meaning the user stays logged in even if the app is closed.
* If rememberMe is false, it sets persistence to SESSION, meaning the user will be logged out when the app is closed.

The **signInWithEmailAndPassword** method is a Firebase Authentication function that attempts to log a user in with an email and password. It’s commonly used in apps to authenticate users based on their credentials.

**\_isLoggedIn = true;**

This line updates the local state variable \_isLoggedIn to true, indicating the user is now logged in.

Example: After successfully entering the club, you get a wristband indicating that you're allowed to stay.

**\_rememberMe = rememberMe;**

This line updates the \_rememberMe variable to reflect the user’s choice about being remembered.

If you decide to tell the club staff (the app) to remember you for next time, they’ll make a note of it.

**SharedPreferences prefs = await SharedPreferences.getInstance();**

This retrieves the instance of SharedPreferences to save user preferences.

Think of it as getting a notepad to jot down important notes about your preferences.

**prefs.setBool('rememberMe', \_rememberMe);**

This line saves the user's "Remember Me" preference in shared preferences.

You write down on your notepad that you want the club to remember you next time.

**notifyListeners();**

Purpose: This line tells any parts of the app that are listening to the AuthProvider that the login state has changed.

It’s like ringing a bell in the club to let everyone know that a new member has arrived, prompting them to adjust the party atmosphere accordingly.

**return null; // Login successful**

This line indicates that the login was successful by returning null.

If you’re given a VIP pass (null means success), it shows that you’re allowed to enjoy the club.

**Logout:**

void logout() async {

    await \_auth.signOut();

    \_isLoggedIn = false;

    SharedPreferences prefs = await SharedPreferences.getInstance();

    prefs.setBool('rememberMe', false);

    print("Remember Me Cleared on Logout");

    notifyListeners();

  }

**Sign Out of Firebase:** await \_auth.signOut(); logs the user out of their Firebase session. This means they are no longer authenticated and don’t have access to any protected areas of the app.

**\_isLoggedIn = false;** sets the app’s internal state to show the user is now logged out.

Clear "Remember Me" Preference:

**SharedPreferences prefs = await SharedPreferences.getInstance();** fetches local storage to access saved preferences.

**prefs.setBool('rememberMe', false);** clears the "Remember Me" setting so that the user won’t stay logged in when the app restarts.

Notify Listeners: notifyListeners(); informs any part of the app that depends on the login status, so the UI can update to reflect that the user is logged out.

**signup**

The signup function is a method to register a new user with Firebase using an email and password.

Future<void> signup(String email, String password) async {

    try {

      await \_auth.createUserWithEmailAndPassword(

          email: email, password: password);

      \_isLoggedIn = true;

      notifyListeners();

    } catch (e) {

      throw "Signup failed: $e";

    }

  }

This line tries to create a new Firebase user with the provided email and password. If successful, Firebase registers the user and automatically logs them in. If there’s an error (like a duplicate email), it will throw an exception.

**SignUpScreen Class**

The SignUpScreen class provides a simple UI for users to register for an account in the app. It includes input fields for the email and password, validates these inputs, and attempts to create a new user through the AuthProvider. If successful, it navigates to the SignUpScreen registration fails, it shows an error message.

// signup\_screen.dart

import 'package:authentiation\_in\_flutter/SignInScreen.dart';

import 'package:flutter/material.dart';

import 'package:provider/provider.dart';

import 'AuthProvider.dart';

class SignUpScreen extends StatelessWidget {

  final \_emailController = TextEditingController();

  final \_passwordController = TextEditingController();

  final \_formKey = GlobalKey<FormState>();

  SignUpScreen({super.key});

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(title: const Text('Sign Up')),

      body: Padding(

        padding: const EdgeInsets.all(16.0),

        child: Form(

          key: \_formKey,

          child: Column(

            children: [

              TextFormField(

                controller: \_emailController,

                decoration: const InputDecoration(labelText: 'Email'),

                validator: (value) => value!.contains('@') ? null : 'Enter a valid email',

              ),

              TextFormField(

                controller: \_passwordController,

                decoration: const InputDecoration(labelText: 'Password'),

                obscureText: true,

                validator: (value) => value!.length >= 6 ? null : 'Password too short',

              ),

        ElevatedButton(

  onPressed: () async {

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

      try {

        await Provider.of<AuthProvider>(context, listen: false)

            .signup(\_emailController.text, \_passwordController.text);

        Navigator.pushReplacement(

          context,

          MaterialPageRoute(builder: (\_) => const SignInScreen()), // or WelcomeScreen()

        );

      } catch (e) {

        ScaffoldMessenger.of(context)

            .showSnackBar(SnackBar(content: Text(e.toString())));

      }

    }

  },

  child: const Text('Sign Up'),

)

            ],

          ),

        ),

      ),

    );

  }

}

**Code Explanation:**

 final \_emailController = TextEditingController();

  final \_passwordController = TextEditingController();

  final \_formKey = GlobalKey<FormState>();

\_emailController and \_passwordController store and manage the input values for email and password.

When you create a **GlobalKey<FormState>** and assign it to a Form widget, you can access the FormState methods. gives you access to the form’s methods, like .validate() for checking field entries. The GlobalKey acts as a unique identifier for the Form. When you assign a GlobalKey to a Form, you gain access to its state—meaning you can manage the Form in a structured way without relying solely on each form field individually.

super.key is used to pass a key parameter from the SignUpScreen constructor to the parent class (StatelessWidget).

In Flutter, Key objects uniquely identify widgets in the widget tree, which helps Flutter efficiently manage and rebuild only parts of the UI that need updates.

**Form widget**

The Form widget is a Flutter widget that provides a convenient way to group multiple input fields for validation and submission.

The Form widget enables validation and helps maintain the form state. Here, it uses the \_formKey to manage its state.

key: \_formKey:The key parameter assigns a unique identifier to the Form widget using \_formKey, which is an instance of GlobalKey<FormState>.

This key allows access to the form's state (like validation status) from other parts of the widget tree. For instance, it allows us to call methods such as \_formKey.currentState!.validate() to validate all fields within the form.

**obscureText: true,**

The obscureText property hides the text as the user types, showing dots instead of characters, to maintain password confidentiality.

**Provider.of<AuthProvider>(context, listen: false):**

The authentication logic (like signing up, logging in, and logging out) is managed in an AuthProvider class. Instead of writing authentication logic inside every screen, we keep it in a central place (the AuthProvider). We need access to this AuthProvider from our SignUp Screen so we can call its signup method when the user clicks the “Sign Up” button.

In a typical Flutter app, widgets are organized in a tree structure. If one widget wants to share its data with another widget that’s deep down in the tree, it can be challenging. **Provider** solves this by creating a central place where data is stored and can be accessed by any widget, regardless of its position in the tree.

The **.of<T>()** method in the Provider package is used to access a specific instance of a provider. Here, AuthProvider is passed as the type argument (<AuthProvider>), meaning we're specifically asking for an instance of AuthProvider.

**(context, ...):** Uses the screen's context to find AuthProvider in the widget tree.

**listen: false** is used when we want to interact with the provider but don’t need to watch for changes.Ideal for calling functions (e.g., login, logout) or for a one-time data access.It improves performance by avoiding unnecessary rebuilds for widgets that don’t need real-time updates.

**Navigator.pushReplacement** is a method in Flutter’s navigation system that changes screens by replacing the current screen with a new one. Unlike Navigator.push, which adds a new screen on top of the current one, pushReplacement removes the current screen from the navigation stack before adding the new one.

The main reason to use **pushReplacement** is when you want to navigate to a new screen and prevent the user from going back to the previous screen using the back button. This is useful in scenarios where returning to the previous screen doesn’t make sense, like after a successful login or signup.

When you move from one screen to another in a Flutter app (like from a signup screen to a login screen), you use a route. **MaterialPageRoute** is a type of route in Flutter that creates a nice sliding transition between screens.

**builder:**

The builder parameter is asking you to define the screen that this route should display. In this case, we’re saying, "When this route is triggered, show SignInScreen."

**(\_) =>:**

This is a shorthand for a function in Dart.

Here, (\_) represents an unused parameter (often called a "throwaway variable"). We use the underscore (\_) to indicate that we don’t need to use it.

The **arrow (=>)** means that whatever follows it is the output or result of this function. In this case, the result is SignInScreen().

**const SignInScreen():**

We’re telling Flutter to navigate to SignInScreen.

Using const means that the SignInScreen widget doesn’t need to change—it’s fixed and won’t rebuild unnecessarily.

**ScaffoldMessenger.of(context):**

This provides a way to show a SnackBar in the app.

ScaffoldMessenger is like a "manager" that controls how messages or alerts are displayed on the screen.

.of(context) finds the current screen where the message should be shown.

**.showSnackBar(SnackBar(...)):**

This displays a SnackBar, which is a small popup message that appears temporarily at the bottom of the screen.

showSnackBar triggers the display of this message.

**SnackBar(content: Text(e.toString())):**

SnackBar takes some content to display. Here, we’re passing Text(e.toString()).

e.toString() converts the error message e into text, so it can be displayed to the user.

**SignInScreen Class:**

class \_SignInScreenState extends State<SignInScreen> {

  final \_emailController = TextEditingController();

  final \_passwordController = TextEditingController();

  final \_formKey = GlobalKey<FormState>();

  final FirebaseAuth \_auth = FirebaseAuth.instance;

  bool \_rememberMe = false;

  String? \_emailError;

  String? \_passwordError;

  String? \_errorMessage;

\_auth provides all the tools needed to manage user authentication, like signing in, signing up, or signing out.

Firebase is like a digital ID verification service at the login window; \_auth is your access to it. When users sign in, you send their credentials to \_auth for validation.

**\_rememberMe** stores the user’s preference about remembering their login for future sessions.

 onChanged: (value) => setState(() => \_rememberMe = value!),

**onChanged:**

* This is an event handler for the Checkbox widget.
* It gets called whenever the user interacts with the checkbox, either checking or unchecking it.

The **value** parameter represents the new state of the checkbox:

* true if the checkbox was just checked.
* false if it was just unchecked.

**(value) =>:**

* This is a shorthand for a one-line anonymous (or inline) function.
* The function takes value as input (the new checkbox state) and runs the setState call.

**setState(() => \_rememberMe = value!):**

* setState tells Flutter that something has changed in the widget, and it needs to rebuild this part of the UI with the new values.
* Inside setState, \_rememberMe is assigned the new value from the checkbox (value).
* The exclamation mark (!) after value is a null check, ensuring that value is not null (useful if working with nullable types).

if (\_errorMessage != null)

                Padding(

                  padding: const EdgeInsets.only(top: 8.0),

                  child: Text(

                    \_errorMessage!,

                    style: const TextStyle(color: Colors.red),

                  ),

                ),

**if (\_errorMessage != null):**

* This checks if \_errorMessage has a non-null value, meaning an error message has been set.
* If \_errorMessage is null, the following code (inside the if statement) won’t be executed, and no error message will be shown.
* If \_errorMessage has a value, then the code inside the if block will run, displaying the error message in the UI.

ElevatedButton(

                onPressed: () async {

                  // Clear previous error messages before each attempt

                  setState(() {

                    \_emailError = null;

                    \_passwordError = null;

                    \_errorMessage = null;

                  });

This part of code is setting up an ElevatedButton with an onPressed function that performs a few specific actions when the button is clicked. Specifically, it’s clearing any previous error messages, so the user sees only new errors after each login attempt.

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

                    // Call login method

                    final errorCode = await authProvider.login(

                      \_emailController.text,

                      \_passwordController.text,

                      \_rememberMe,

                    );

                    setState(() {

                      if (errorCode == null) {

                        // Login successful: navigate to WelcomeScreen

                        Navigator.pushReplacement(

                          context,

                          MaterialPageRoute(builder: (\_) => const WelcomeScreen()),

                        );

                      } else {

                        // Set generic error message for invalid credentials

                        \_errorMessage = "Invalid credentials. Please try again.";

                      }

                    });

                  }

**\_formKey.currentState!.validate()** runs the validator method of each TextFormField in the form. If any field fails validation (e.g., email format is incorrect, or the password is too short), this condition will return false, and the code block will be skipped.

**the login function in authProvider and awaits its result.**

It passes three parameters:

* \_emailController.text: The email entered by the user.
* \_passwordController.text: The password entered by the user.
* \_rememberMe: The user's preference to stay logged in.

login returns null if the login is successful or an error code if it fails.

**setState** is called to update the UI based on the login attempt's outcome.

* If errorCode is null, it means the login was successful.
* The app then navigates to the WelcomeScreen, replacing the current screen with it. This is done using Navigator.pushReplacement, which removes the current screen from the stack so the user cannot navigate back to it with the back button.
* MaterialPageRoute is used to create a route to the WelcomeScreen.

**Main Class**

**WidgetsFlutterBinding.ensureInitialized();**

* This line ensures that Flutter’s engine is initialized before the app does anything else.
* It’s necessary here because Firebase initialization requires Flutter to be fully ready (especially in an async context).

 await Firebase.initializeApp(

    options: const FirebaseOptions(

      apiKey: 'AIzaSyC2ulWD3tpV2zXecGXC\_nzq3qkxPjmO\_NQ',

      appId: '1:543871071691:android:df3e42d0f5fb671e77b24e',

      messagingSenderId: '543871071691',

      projectId: 'authentication-72ad7',

    ),

  );

This line initializes Firebase, which is required before using any Firebase services.

await ensures that Firebase is completely initialized before proceeding.

* FirebaseOptions provides the necessary configuration values:
* **apiKey:** Unique identifier for accessing Firebase services.
* **appId:** Identifies the app within the Firebase project.
* **messagingSenderId:** Identifies the app’s sender ID for Firebase Cloud Messaging.
* **projectId:** Specifies the Firebase project.

**runApp(...):**

This function starts your Flutter app. It takes a widget as an argument, which is usually the root widget of the app. Here, we’re passing in ChangeNotifierProvider as the root widget instead of directly passing MyApp.

Wrapping MyApp in **ChangeNotifierProvider** means that every widget within MyApp (and its child widgets) will have access to AuthProvider.

This setup allows widgets anywhere inside MyApp to access authentication information and actions from AuthProvider without needing to pass it around manually.

If we didn’t use **ChangeNotifierProvider**, it would be difficult to keep track of authentication state (like logged-in status) across different parts of the app. By using ChangeNotifierProvider, we can:

Easily access and modify authentication state (like login and logout) anywhere in the app.

Automatically notify parts of the app that depend on AuthProvider whenever its data changes (for example, if a user logs in or out).

wrapping MyApp in ChangeNotifierProvider, we’re making AuthProvider accessible across the app and ensuring the app stays up-to-date with authentication status changes.

**create: (context) => AuthProvider(),:**

This line tells ChangeNotifierProvider to create an instance of AuthProvider when the app starts.

**Authorization in Flutter**

**1. User Authentication**

**Sign Up:** Users can create accounts using email and password.

**Sign In:** Users can log in with their credentials (email and password). Admins have hardcoded credentials for easy access.

**Role-based Authentication:** Based on the role (Admin, Employee, or Customer), users are redirected to different dashboards.

**2. Role-based Dashboards**

**Admin Dashboard:**

* Admins can log in and view their dashboard.
* Admins can add new employees by providing an email and password.
* Admins can log out from the app.

**Employee Dashboard:**

* Employees see a different dashboard where they can manage tasks or profile (functionality can be added later).
* Employees can log out.

**Customer Dashboard:**

* Customers have a separate dashboard to manage their orders (functionality can be added later).
* Customers can log out.

**3. Navbar (Navigation Bar)**

**Common Navbar:** We created a common navigation bar that adjusts based on the user’s role:

* Admin sees options like managing employees and other admin-related tasks.
* Employee sees options relevant to their role (e.g., tasks).
* Customer sees options like viewing orders.

**Visibility of Options:** Depending on the role (Admin, Employee, or Customer), some options are hidden from the navbar.

**4. Logout Functionality**

Logout: All users (Admin, Employee, Customer) can log out from their dashboards, which will take them back to the sign-in page.

**5. Firestore Integration**

**User Data:** The role of each user is stored in Firestore. When a user logs in, the app fetches the role from Firestore and redirects them to the appropriate dashboard.

**Dependencies:**

Add following dependencies in your project:

  firebase\_core: ^3.8.0

  firebase\_auth: ^5.3.3

  cloud\_firestore: ^5.5.0

**Configure and attach with firebase database**

**Signup Class:**

import 'package:flutter/material.dart';

import 'package:firebase\_auth/firebase\_auth.dart';

import 'package:cloud\_firestore/cloud\_firestore.dart';

class SignupPage extends StatefulWidget {

  @override

  \_SignupPageState createState() => \_SignupPageState();

}

class \_SignupPageState extends State<SignupPage> {

  final FirebaseAuth \_auth = FirebaseAuth.instance;

  final FirebaseFirestore \_firestore = FirebaseFirestore.instance;

  final TextEditingController \_emailController = TextEditingController();

  final TextEditingController \_passwordController = TextEditingController();

  bool \_isLoading = false;

  void \_signup() async {

    setState(() {

      \_isLoading = true;

    });

    try {

      // Create user with email and password

      UserCredential userCredential = await \_auth.createUserWithEmailAndPassword(

        email: \_emailController.text.trim(),

        password: \_passwordController.text.trim(),

      );

      // Write user data to Firestore

      await \_firestore.collection('users').doc(userCredential.user!.uid).set({

        'email': \_emailController.text.trim(),

        'role': 'Customer', // Default role for signup

      });

      // Navigate to AuthPage

      Navigator.pushReplacementNamed(context, '/');

    } catch (e) {

      ScaffoldMessenger.of(context).showSnackBar(

        SnackBar(content: Text(e.toString())),

      );

    } finally {

      setState(() {

        \_isLoading = false;

      });

    }

  }

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(title: Text('Sign Up')),

      body: Padding(

        padding: const EdgeInsets.all(16.0),

        child: Column(

          crossAxisAlignment: CrossAxisAlignment.stretch,

          children: [

            TextField(

              controller: \_emailController,

              decoration: InputDecoration(labelText: 'Email'),

            ),

            TextField(

              controller: \_passwordController,

              decoration: InputDecoration(labelText: 'Password'),

              obscureText: true,

            ),

            SizedBox(height: 16),

            \_isLoading

                ? Center(child: CircularProgressIndicator())

                : ElevatedButton(

                    onPressed: \_signup,

                    child: Text('Sign Up'),

                  ),

          ],

        ),

      ),

    );

  }

}

**Explanation Of class:**

class SignupPage extends StatefulWidget {

  @override

  \_SignupPageState createState() => \_SignupPageState();

}

**SignupPage** is a StatefulWidget because the UI changes based on user interaction (e.g., loading state, form submission).

**createState()** creates the state object \_SignupPageState, which manages the UI and logic of the SignupPage.

class \_SignupPageState extends State<SignupPage> {

  final FirebaseAuth \_auth = FirebaseAuth.instance;

  final FirebaseFirestore \_firestore = FirebaseFirestore.instance;

  final TextEditingController \_emailController = TextEditingController();

  final TextEditingController \_passwordController = TextEditingController();

  bool \_isLoading = false;

  void \_signup() async {

    setState(() {

      \_isLoading = true;

    });

\_auth: This is an instance of FirebaseAuth, which is used to handle authentication actions like user sign-up.

\_firestore: This is an instance of FirebaseFirestore, which is used to interact with Firestore (storing user data).

\_emailController and \_passwordController: These are TextEditingController objects that are used to control the input fields for email and password.

\_isLoading: This boolean flag tracks whether the app is in a loading state (i.e., when the sign-up process is happening).

\_signup() is an asynchronous method that handles the user sign-up process.

setState() is called to update the \_isLoading state to true, which will trigger the UI to show a loading indicator.

 try {

      // Create user with email and password

      UserCredential userCredential = await \_auth.createUserWithEmailAndPassword(

        email: \_emailController.text.trim(),

        password: \_passwordController.text.trim(),

      );

The **UserCredential** class in Firebase is like a receipt that you get when a user signs in, signs up, or re-authenticates. This receipt contains three main things:

* Who is the user?
* Was the account just created, or is it an existing one?
* What credentials were used?

**When a user signs up**

Firebase creates a new account for the user, and UserCredential gives you the details of the new account.

**When a user logs in**

Firebase confirms the login, and UserCredential gives you the details of the logged-in user.

**Creates a New User in Firebase Authentication:**

* It uses the createUserWithEmailAndPassword method from Firebase to create a new user account with the provided email and password.
* The email and password are retrieved from the user input (\_emailController and \_passwordController).

**Trims Input:**

.trim() ensures there are no unnecessary spaces before or after the email or password.

Captures the User Information:

* The result of this operation is stored in a UserCredential object (userCredential).
* This object contains details about the newly created user, such as their unique Firebase ID (uid) and email address.

**Handles Asynchronous Behavior:**

* The await keyword ensures the app waits for Firebase to finish creating the account before moving to the next line of code.

 await \_firestore.collection('users').doc(userCredential.user!.uid).set({

        'email': \_emailController.text.trim(),

        'role': 'Customer', // Default role for signup

      });

This line of code stores the newly created user's email and role ('Customer') in the Firestore database under the users collection. The document is uniquely identified by the user's Firebase Authentication uid. This ensures that user data is properly stored and can be retrieved later.

**.doc(userCredential.user!.uid):** It creates a new document with the unique uid of the newly created user (from the userCredential object).

the **userCredential.user!.uid** is the unique identifier for the user, which was generated when the user was created in Firebase Authentication.

**.set({}):** This method is used to set or update the data for the user document.

Inside the set() method, two fields are set:

**'email':** The email address of the user, which is retrieved from the sign-up form (\_emailController.text.trim()).

**'role':** A role is assigned to the user. In this case, the default role is set to 'Customer' for all new users who sign up.

**userCredential.user!.uid:**

* userCredential.user is the User object, and uid is a property of this User object.
* uid is a unique identifier for the user assigned by Firebase Authentication. This uid will never change, and it can be used to uniquely identify the user in the Firebase system.
* The ! after user is a null check operator. It tells Dart that you're sure the user object is not null at this point. In other words, after a successful sign-up, Firebase guarantees that the user object will not be null.

**.doc(userCredential.user!.uid):**

* .doc() is a method from Firestore that refers to a specific document.
* By passing userCredential.user!.uid as the argument to .doc(), you're telling Firestore to create or access a document with the ID that matches the user's unique uid.
* Essentially, you're creating a document in Firestore for the user where the document ID is their unique uid.
* user refers to the User object that contains information about the authenticated user.
* user.uid gives the unique identifier (UID) of that user, which is a string that Firebase uses to uniquely identify the user.

**pushReplacementNamed:**

* This method is used to navigate to a new route and replace the current route in the navigation stack.
* When you use pushReplacementNamed, the current screen is removed from the stack and replaced by the new screen. This means the user will not be able to press the back button to return to the previous screen.

**context:**

* context refers to the current location in the widget tree. It’s used to get information about the widget’s location and to interact with other widgets (like Navigator).
* The context is necessary for navigation because it helps the Navigator know where it is in the widget tree.

**'/':**

* The string '/' is the route name that you want to navigate to.
* In Flutter, you can define routes in the MaterialApp widget using the routes parameter. The route '/' is often used as the default route or the home screen of the app.
* If you navigate to '/', it will typically take the user to the home screen of the app, unless you’ve defined the route differently.

catch (e) {

      ScaffoldMessenger.of(context).showSnackBar(

        SnackBar(content: Text(e.toString())),

      );

    } finally {

      setState(() {

        \_isLoading = false;

      });

    }

  }

* catch (e): Catches any error that occurs during the signup process.
* ScaffoldMessenger.of(context).showSnackBar(...): Displays the error message in a SnackBar at the bottom of the screen.
* SnackBar(content: Text(e.toString())): Converts the error (e) into a string and shows it in the SnackBar.
* finally: Ensures that the code inside it runs after the try and catch blocks, regardless of success or failure.
* setState(() { \_isLoading = false; }): Stops the loading spinner by setting \_isLoading to false.

**Sign In Page:**

import 'package:flutter/material.dart';

import 'package:firebase\_auth/firebase\_auth.dart';

import 'package:cloud\_firestore/cloud\_firestore.dart';

import 'auth\_page.dart';

import 'package:role\_base\_system/signup.dart'; // Import your signup page

class SignInPage extends StatefulWidget {

  @override

  \_SignInPageState createState() => \_SignInPageState();

}

class \_SignInPageState extends State<SignInPage> {

  final FirebaseAuth \_auth = FirebaseAuth.instance;

  final FirebaseFirestore \_firestore = FirebaseFirestore.instance;

  final TextEditingController \_emailController = TextEditingController();

  final TextEditingController \_passwordController = TextEditingController();

  bool \_isLoading = false;

  void \_signIn() async {

    setState(() {

      \_isLoading = true;

    });

    try {

      // Admin hardcoded credentials for testing

      if (\_emailController.text == 'admin@example.com' &&

          \_passwordController.text == 'admin123') {

        Navigator.pushReplacementNamed(context, '/admin\_dashboard');

        return;

      }

      // Sign in with Firebase Auth

      UserCredential userCredential = await \_auth.signInWithEmailAndPassword(

        email: \_emailController.text.trim(),

        password: \_passwordController.text.trim(),

      );

      // Fetch user role from Firestore

      DocumentSnapshot userDoc = await \_firestore

          .collection('users')

          .doc(userCredential.user!.uid)

          .get();

      if (userDoc.exists) {

        String role = userDoc.get('role');

        String email = userDoc.get('email'); // Fetch email from Firestore

        // Navigate based on role

        if (role == 'Admin') {

          Navigator.pushReplacementNamed(context, '/admin\_dashboard');

        } else if (role == 'Employee') {

          Navigator.pushReplacementNamed(context, '/employee\_dashboard');

        } else if (role == 'Customer') {

          Navigator.pushReplacementNamed(context, '/customer\_dashboard');

        } else {

          ScaffoldMessenger.of(context).showSnackBar(

            SnackBar(content: Text('Unknown role. Please contact admin.')),

          );

        }

      } else {

        ScaffoldMessenger.of(context).showSnackBar(

          SnackBar(content: Text('User data not found in Firestore.')),

        );

      }

    } catch (e) {

      ScaffoldMessenger.of(context).showSnackBar(

        SnackBar(content: Text('Error: ${e.toString()}')),

      );

    } finally {

      setState(() {

        \_isLoading = false;

      });

    }

  }

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      appBar: AppBar(title: Text('Sign In')),

      body: Padding(

        padding: const EdgeInsets.all(16.0),

        child: Column(

          crossAxisAlignment: CrossAxisAlignment.stretch,

          children: [

            TextField(

              controller: \_emailController,

              decoration: InputDecoration(labelText: 'Email'),

            ),

            TextField(

              controller: \_passwordController,

              decoration: InputDecoration(labelText: 'Password'),

              obscureText: true,

            ),

            SizedBox(height: 16),

            \_isLoading

                ? Center(child: CircularProgressIndicator())

                : ElevatedButton(

                    onPressed: \_signIn,

                    child: Text('Sign In'),

                  ),

            SizedBox(height: 16),

            Row(

              mainAxisAlignment: MainAxisAlignment.center,

              children: [

                Text("Don't have an account? "),

                GestureDetector(

                  onTap: () {

                    Navigator.push(

                      context,

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

                    );

                  },

                  child: Text(

                    'Create Account',

                    style: TextStyle(

                      color: Colors.blue,

                      fontWeight: FontWeight.bold,

                    ),

                  ),

                ),

              ],

            ),

          ],

        ),

      ),

    );

  }

}

A **DocumentSnapshot** represents the data and metadata of a single document in a Firestore database.

* It retrieves the data from a specific document.
* It lets you access fields (like name, email, etc.) stored in the document.
* It tells you if the document exists or not.
* In short: It’s like a snapshot of a single document's content from the Firestore database.

**ScaffoldMessenger.of(context):** Finds the current ScaffoldMessenger for the screen where the snack bar will appear.

**.showSnackBar(...):** Displays the snack bar using the found ScaffoldMessenger.

**SnackBar(content: Text(...)):** Defines the snack bar's content as a message to inform the user of the issue ("Unknown role. Please contact admin.").

This entire snippet ensures that a user is notified with a temporary message when their role is unrecognized. The snack bar disappears automatically after a short duration.

**AuthPage:**

AuthPage ensures that authenticated users are routed correctly to their respective dashboards and handles edge cases, making it an essential component of the app's flow. When a user opens the app, they may already be logged in from a previous session.AuthPage checks if the user is authenticated and automatically redirects them to the correct dashboard without requiring them to log in again.

*Example:*

**SignInPage:** When you first open the app, you log in with your credentials.

**AuthPage:** After logging in, Netflix remembers your session and directly shows your profile/dashboard when you reopen the app. If your session expires, it redirects you back to the login screen.

class AuthPage extends StatelessWidget {

  final FirebaseAuth \_auth = FirebaseAuth.instance;

  final FirebaseFirestore \_firestore = FirebaseFirestore.instance;

  @override

  Widget build(BuildContext context) {

    return Scaffold(

      body: StreamBuilder<User?>(

        stream: \_auth.authStateChanges(),

        builder: (context, snapshot) {

          if (snapshot.connectionState == ConnectionState.waiting) {

            return Center(child: CircularProgressIndicator());

          } else if (snapshot.hasData) {

            String uid = snapshot.data!.uid;

            return FutureBuilder<DocumentSnapshot>(

              future: \_firestore.collection('users').doc(uid).get(),

              builder: (context, snapshot) {

                if (snapshot.connectionState == ConnectionState.waiting) {

                  return Center(child: CircularProgressIndicator());

                } else if (snapshot.hasData && snapshot.data!.exists) {

                  String role = snapshot.data!.get('role');

                  String email = snapshot.data!.get('email'); // Access email here

                  if (role == 'Admin') {

                    return AdminDashboard();

                  } else if (role == 'Employee') {

                    return EmployeeDashboard();

                  } else if (role == 'Customer') {

                    return CustomerDashboard();

                  } else {

                    return Center(child: Text('Unknown role. Please contact admin.'));

                  }

                } else {

                  return Center(

                    child: Column(

                      mainAxisAlignment: MainAxisAlignment.center,

                      children: [

                        Text(

                          'User data not found. Please try again.',

                          style: TextStyle(fontSize: 16),

                        ),

                        SizedBox(height: 20),

                        ElevatedButton(

                          onPressed: () {

                            FirebaseAuth.instance.signOut();

                            Navigator.pushReplacementNamed(context, '/signin');

                          },

                          child: Text('Sign Out'),

                        ),

                      ],

                    ),

                  );

                }

              },

            );

          } else {

            return SignInPage();

          }

        },

      ),

    );

  }

}

\_auth: Handles Firebase Authentication (e.g., login/logout, tracking user state).

\_firestore: Connects to Firestore to fetch user data (e.g., role, email).

The **StreamBuilder** class in Flutter is a widget that listens to a stream of data and rebuilds its UI whenever the stream emits new data. It is widely used for handling asynchronous data streams, such as real-time updates from Firebase, user authentication state changes, or WebSocket connections.

The **User** class in Firebase Authentication provides details about the currently signed-in user. It contains properties and methods to access user-specific information and perform user-related actions.

The User object is nullable (User?) because:

* A user might not be signed in, in which case the value will be null.
* The authStateChanges stream emits null when the user logs out.

**authStateChanges()** is a stream provided by Firebase Authentication that listens to changes in the user's authentication state. It notifies you whenever the user's authentication status changes. This could happen when:

* A user signs in (e.g., using email/password, Google sign-in, etc.).
* A user signs out.
* A user’s authentication state is updated (e.g., their session expires or they are reauthenticated).

 builder: (context, snapshot) {

          if (snapshot.connectionState == ConnectionState.waiting) {

            return Center(child: CircularProgressIndicator());

          }

**builder: (context, snapshot):**

The builder is a callback function provided by the StreamBuilder widget.It takes two parameters:

**context:** The context of the widget (which is necessary for accessing other widgets and services).

**snapshot:** This contains the data or state of the stream. In this case, it holds the state of the authentication stream.

When a user opens the app, Firebase checks whether the user is logged in or not. During this check, the app waits for the result. This code ensures that while waiting for Firebase to provide the result, a loading spinner is displayed to the user. Once Firebase responds (either the user is signed in or not), the UI will update accordingly.

 else if (snapshot.hasData) {

            String uid = snapshot.data!.uid;

            return FutureBuilder<DocumentSnapshot>(

              future: \_firestore.collection('users').doc(uid).get(),

              builder: (context, snapshot) {

                if (snapshot.connectionState == ConnectionState.waiting) {

                  return Center(child: CircularProgressIndicator());

                } else if (snapshot.hasData && snapshot.data!.exists) {

                  String role = snapshot.data!.get('role');

                  String email = snapshot.data!.get('email'); // Access email here

                  if (role == 'Admin') {

                    return AdminDashboard();

                  } else if (role == 'Employee') {

                    return EmployeeDashboard();

                  } else if (role == 'Customer') {

                    return CustomerDashboard();

                  } else {

                    return Center(child: Text('Unknown role. Please contact admin.'));

                  }

                } else {

                  return Center(

                    child: Column(

                      mainAxisAlignment: MainAxisAlignment.center,

                      children: [

                        Text(

                          'User data not found. Please try again.',

                          style: TextStyle(fontSize: 16),

                        ),

                        SizedBox(height: 20),

                        ElevatedButton(

                          onPressed: () {

                            FirebaseAuth.instance.signOut();

                            Navigator.pushReplacementNamed(context, '/signin');

                          },

                          child: Text('Sign Out'),

                        ),

                      ],

                    ),

                  );

                }

              },

            );

          }

**else if (snapshot.hasData):**

This checks if the snapshot from the StreamBuilder has data. If the user is authenticated (i.e., logged in), snapshot.hasData will be true.If the user is logged in, the code inside this block will execute.

**String uid = snapshot.data!.uid;:**

Here, we are getting the unique ID (uid) of the authenticated user from the snapshot.data.

snapshot.data! refers to the User object that contains the user's details (like uid).

**return FutureBuilder<DocumentSnapshot>(...):**

* We use a FutureBuilder here to fetch the user's data from Firestore.
* The FutureBuilder listens for a future (in this case, fetching the document from Firestore) and updates the UI when the future completes.
* The future parameter is the Firestore query that gets the user document from the users collection using the uid of the logged-in user.

**builder: (context, snapshot):**

* This builder is a callback function that is triggered whenever the state of the FutureBuilder changes.
* It takes two parameters: context (the context of the widget) and snapshot (the result of the future, which is the user's Firestore document).

**if (snapshot.connectionState == ConnectionState.waiting):**

This checks if the FutureBuilder is still waiting for the data from Firestore.

If the data is still loading, it shows a loading spinner (CircularProgressIndicator) to indicate that the app is waiting for the result.

**else if (snapshot.hasData && snapshot.data!.exists):**

* Once the data has been loaded, it checks if the document exists (snapshot.data!.exists).
* If the document exists, it retrieves the role and email from the Firestore document.

**String role = snapshot.data!.get('role');:**

This retrieves the role field from the Firestore document. The role determines whether the user is an "Admin", "Employee", or "Customer".

**String email = snapshot.data!.get('email');:**

This retrieves the email field from the Firestore document. You can use the email for other purposes if needed.

**if (role == 'Admin') { return AdminDashboard(); }:**

* Based on the role, the user is directed to a specific dashboard:
* If the role is "Admin", it navigates to the AdminDashboard.

.

**else { return Center(child: Text('Unknown role. Please contact admin.')); }:**

If the role is not one of the predefined roles ("Admin", "Employee", "Customer"), it shows a message indicating that the role is unknown and asks the user to contact the admin.

**else { return Center(child: Column(...)); }:**

If the document does not exist (meaning there was an issue fetching the user data), it displays an error message: "User data not found. Please try again."

It also provides a "Sign Out" button, which signs the user out and redirects them to the sign-in page (/signin).

**Common Navbar**

import 'package:flutter/material.dart';

class CommonNavbar extends StatelessWidget {

  final String role;

  CommonNavbar({required this.role});

  @override

  Widget build(BuildContext context) {

    return Drawer(

      child: ListView(

        padding: EdgeInsets.zero,

        children: [

          DrawerHeader(

            decoration: BoxDecoration(color: Colors.blue),

            child: Text(

              'Welcome, $role',

              style: TextStyle(color: Colors.white, fontSize: 24),

            ),

          ),

          // Common option for all roles

          ListTile(

            leading: Icon(Icons.home),

            title: Text('Home'),

            onTap: () {

              Navigator.pushNamed(context, '/home');

            },

          ),

          // Admin-only option

          if (role == 'Admin')

            ListTile(

              leading: Icon(Icons.person\_add),

              title: Text('Add Employee'),

              onTap: () {

                Navigator.pushNamed(context, '/add\_employee');

              },

            ),

          // Employee-only option

          if (role == 'Employee')

            ListTile(

              leading: Icon(Icons.task),

              title: Text('View Tasks'),

              onTap: () {

                Navigator.pushNamed(context, '/tasks');

              },

            ),

          // Customer-only option

          if (role == 'Customer')

            ListTile(

              leading: Icon(Icons.shopping\_cart),

              title: Text('Shop'),

              onTap: () {

                Navigator.pushNamed(context, '/shop');

              },

            ),

          // Logout option (common for all roles)

          ListTile(

            leading: Icon(Icons.logout),

            title: Text('Logout'),

            onTap: () {

              Navigator.pushNamed(context, '/signin');

            },

          ),

        ],

      ),

    );

  }

}

A CommonNavbar widget in Flutter, which provides a side navigation drawer with different menu options based on the user's role (Admin, Employee, or Customer). Here's a breakdown of how it works:

class CommonNavbar extends StatelessWidget {

final String role;

CommonNavbar({required this.role});

* This class is a StatelessWidget because the content of the navigation bar doesn't change dynamically after the initial build.
* The class takes a role parameter, which indicates the user's role (Admin, Employee, or Customer). This role is passed when creating the CommonNavbar widget.

**Drawer Widget:**

return Drawer(

child: ListView(

padding: EdgeInsets.zero,

children: [

* The main structure of the navigation drawer is created using the Drawer widget, which is a slide-in menu typically used for app navigation.
* Inside the Drawer, we use a ListView to display a list of options.

**DrawerHeader:**

DrawerHeader(

decoration: BoxDecoration(color: Colors.blue),

child: Text(

'Welcome, $role',

style: TextStyle(color: Colors.white, fontSize: 24),

),

),

* The DrawerHeader provides a welcome message at the top of the drawer. It uses the role passed to the widget to display a personalized greeting (e.g., "Welcome, Admin").
* The header has a blue background with white text.

**Common Menu Option (for all roles):**

ListTile(

leading: Icon(Icons.home),

title: Text('Home'),

onTap: () {

Navigator.pushNamed(context, '/home');

},

),

* This option is common for all roles and leads to the home screen. It has a home icon and a label "Home".
* When the user taps on this option, it navigates to the /home route using Navigator.pushNamed.

**Role-Specific Menu Options:**

**Admin-only Option:**

if (role == 'Admin')

ListTile(

leading: Icon(Icons.person\_add),

title: Text('Add Employee'),

onTap: () {

Navigator.pushNamed(context, '/add\_employee');

},

),

If the user's role is "Admin", they will see an option to "Add Employee", which navigates to the /add\_employee route.

**Employee-only Option:**

if (role == 'Employee')

ListTile(

leading: Icon(Icons.task),

title: Text('View Tasks'),

onTap: () {

Navigator.pushNamed(context, '/tasks');

},

),

If the user's role is "Employee", they will see an option to "View Tasks", which navigates to the /tasks route.

**Customer-only Option:**

if (role == 'Customer')

ListTile(

leading: Icon(Icons.shopping\_cart),

title: Text('Shop'),

onTap: () {

Navigator.pushNamed(context, '/shop');

},

),

If the user's role is "Customer", they will see an option to "Shop", which navigates to the /shop route.

**Logout Option (Common for all roles):**

ListTile(

leading: Icon(Icons.logout),

title: Text('Logout'),

onTap: () {

Navigator.pushNamed(context, '/signin');

},

),

* This is a common logout option for all roles. It has a logout icon and label "Logout".
* When tapped, it navigates to the /signin route, which presumably shows the sign-in page for the user to log out.

void \_logout(BuildContext context) async {

    bool confirm = await showDialog(

      context: context,

      builder: (context) {

        return AlertDialog(

          title: Text('Confirm Logout'),

          content: Text('Are you sure you want to log out?'),

          actions: [

            TextButton(

              onPressed: () => Navigator.pop(context, false),

              child: Text('Cancel'),

            ),

            ElevatedButton(

              onPressed: () => Navigator.pop(context, true),

              child: Text('Logout'),

            ),

          ],

        );

      },

    );

    if (confirm) {

      await \_auth.signOut();

      Navigator.pushReplacementNamed(context, '/signin');

    }

  }

**The Navigator.pop(context)** function in Flutter is used to pop or remove the current screen (or route) from the navigation stack and return to the previous screen.

We put all our code in Boolean variable “confirm”, it will set true or false depends on .pop return.

**showDialog:** This function opens a dialog asking the user to confirm if they want to log out. The dialog has two buttons: Cancel and Logout.

**Navigator.pop(context, false):** When the user presses Cancel, the dialog closes and returns false to the previous screen. This means the user did not want to log out.

**Navigator.pop(context, true):** When the user presses Logout, the dialog closes and returns true to the previous screen. This means the user wants to log out.

**await showDialog(...):** The await ensures that the dialog is shown and we wait for the user to respond (either by pressing Cancel or Logout).

**if (confirm):** After the dialog is closed, we check the returned value (true or false).

If confirm is true (meaning the user clicked Logout), we proceed to sign the user out and navigate to the sign-in page.

If confirm is false (meaning the user clicked Cancel), nothing happens, and the user remains on the current screen.

  User? currentUser = \_auth.currentUser;

**currentUser:** This is a property of the FirebaseAuth class. It gives you the currently logged-in user, or null if no user is logged in. The currentUser property returns an object of type User?, which means it can either be a User object or null.

**drawer: CommonNavbar(role: 'Admin')**

The **CommonNavbar** widget is designed to display a drawer with different options based on the user's role (e.g., Admin, Employee, Customer). In this case, since the role: 'Admin' is passed, the drawer will likely show options relevant to an Admin, such as managing employees, viewing admin-specific tasks, etc.

**Image Uploading (CRUD)**

**Add Product details in Firebase**

class AddProductPage extends StatefulWidget {

  const AddProductPage({Key? key}) : super(key: key);

  @override

  \_AddProductPageState createState() => \_AddProductPageState();

}

* A StatefulWidget is like a part of your app that can change over time.
* For example, if you have a form where users can type a product name, upload an image, and press "Add Product," that form changes as users interact with it.
* AddProductPage is a StatefulWidget because it needs to update when the user types in the form or picks an image.

You’re building a screen in a Flutter app where users can add a new product. This screen:

* Shows a form to type the product name, price, and quantity.
* Lets users upload an image for the product.
* Saves the product details when the "Add Product" button is clicked.

**const AddProductPage({Key? key}) : super(key: key);**

This is the constructor for your screen. Think of it as the "entry point" for creating this screen.

**const:**

This makes the screen "fixed" or unchangeable if nothing inside it changes. It helps Flutter run faster.

**Key? key:**

This is like a "name tag" for the screen. It helps Flutter keep track of this screen if there are many screens in your app. This declares a variable named key of **type Key? (a nullable Key).**

It’s like saying: “This widget can optionally accept a Key.”

* Second key is the name of the parameter being passed to the constructor.
* First key: Declares the parameter (a Key? that the widget can accept).
* Second key: The actual name of the parameter (used when passing a value).
* key is not a keyword.
* Yes, you can replace key with any other word, like uniqueKey or id.
* But it’s better to stick with key for consistency and compatibility with Flutter's conventions.

**: super(key: key)** takes the key from your widget’s constructor and sends it to the parent class (StatefulWidget or StatelessWidget).

* The parent class (StatefulWidget or StatelessWidget) uses this key to:
* Identify this widget uniquely in the widget tree.
* Help Flutter efficiently rebuild or update widgets.

The first key is the parameter name in the parent class (StatefulWidget or StatelessWidget).

The second key is the value passed to your widget’s constructor.

Uint8List? \_selectedImage;

  final TextEditingController \_nameController = TextEditingController();

  final TextEditingController \_priceController = TextEditingController();

  final TextEditingController \_quantityController = TextEditingController();

  Future<void> \_pickImage() async {

    final ImagePicker picker = ImagePicker();

    final XFile? pickedFile = await picker.pickImage(source: ImageSource.gallery);

    if (pickedFile != null) {

      final bytes = await pickedFile.readAsBytes();

      setState(() {

        \_selectedImage = bytes;

      });

    }

  }

* **Uint8List** is used to store binary data (e.g., an image file in memory as bytes).
* The ? makes it nullable, meaning \_selectedImage can either store an image or be null if no image is selected.
* This variable will hold the image data after the user selects an image from their gallery.

**ImagePicker** is a Flutter package used to pick images or videos from the device’s gallery or camera.

You need an ImagePicker instance to access the device's gallery or camera.

**final XFile? pickedFile = await picker.pickImage(source: ImageSource.gallery);**

* image picker return xfile type data so we are making variable of type xfile . This is a result object (class) returned by ImagePicker when a user selects a file. It contains information about the selected file (like its path or content).
* After you select a file, the file manager hands you the file object (XFile), which contains details about the file (e.g., path, size).
* The picker.pickImage method opens the gallery.
* If the user selects an image, it returns an XFile object.
* The path property gives the file's location.
* The readAsBytes method reads the image as binary data.
* If no file is selected, pickedFile is null.

The **source: ImageSource.gallery** is a parameter passed to the pickImage method of the ImagePicker class in Flutter. It specifies where the image should be picked from.

**The ImageSource enum has two possible values:**

**ImageSource.gallery:**

Opens the device’s photo gallery, allowing the user to select an image already stored on their device.

**ImageSource.camera:**

Opens the device’s camera, allowing the user to take a new photo.

void \_addProduct() async {

    final name = \_nameController.text;

    final price = double.tryParse(\_priceController.text);

    final quantity = int.tryParse(\_quantityController.text);

    if (name.isEmpty || price == null || quantity == null || \_selectedImage == null) {

      ScaffoldMessenger.of(context).showSnackBar(

        const SnackBar(content: Text('Please fill all fields and select an image')),

      );

      return;

    }

    // Convert image to Base64

    final imageBase64 = base64Encode(\_selectedImage!);

    // Create a product map

    final product = {

      'name': name,

      'price': price,

      'quantity': quantity,

      'image': imageBase64,

    };

    try {

      // Add product to Firestore

      await FirebaseFirestore.instance.collection('products').add(product);

      // Show success message

      ScaffoldMessenger.of(context).showSnackBar(

        const SnackBar(content: Text('Product added successfully')),

      );

      // Clear inputs and image

      setState(() {

        \_nameController.clear();

        \_priceController.clear();

        \_quantityController.clear();

        \_selectedImage = null;

      });

      // Optionally navigate back to the product list

      Navigator.pop(context);

    } catch (e) {

      ScaffoldMessenger.of(context).showSnackBar(

        SnackBar(content: Text('Error adding product: $e')),

      );

    }

  }

*Firestore doesn’t support storing raw binary data directly, so the image is converted to a format that can be stored as a string.*

**final imageBase64 = base64Encode(\_selectedImage!);**

* Converts the selected image (stored as raw bytes in \_selectedImage) into a Base64 string.
* Base64 is a text-based representation of binary data, which is easier to store in databases like Firestore.

// Create a product map

    final product = {

      'name': name,

      'price': price,

      'quantity': quantity,

      'image': imageBase64,

    };

    try {

      // Add product to Firestore

      await FirebaseFirestore.instance.collection('products').add(product);

      // Show success message

      ScaffoldMessenger.of(context).showSnackBar(

        const SnackBar(content: Text('Product added successfully')),

      );

* a map is a data structure that stores key-value pairs.
* When adding a product, we need to send multiple pieces of information (name, price, quantity, image).
* A map bundles all this information together into one object that can be passed around easily.
* Without a map, you would need to handle each piece of data individually, which is inefficient and error-prone.
* Firestore databases are designed to work with structured data.
* Firestore collections (like products) store documents, and each document is essentially a map.
* By creating a product map, you align your data with Firestore’s expected format, making it easy to add to the database:

**Navigator.pop(context);**

Navigator.pop(context) tells the app to close the current screen (the "Add Product" page) and return to the previous screen (likely the product list page). pop(context) removes the top screen (current screen) from the stack, revealing the previous screen underneath.

TextField(

                controller: \_priceController,

                decoration: const InputDecoration(labelText: 'Product Price'),

                keyboardType: TextInputType.number,

              ),

keyboardType (for price and quantity fields): Ensures the correct keyboard (numeric) appears.

 Center(

                child: GestureDetector(

                  onTap: \_pickImage,

                  child: \_selectedImage != null

                      ? Container(

                          width: 150,

                          height: 150,

                          decoration: BoxDecoration(

                            border: Border.all(color: Colors.grey),

                            borderRadius: BorderRadius.circular(10),

                          ),

                          child: ClipRRect(

                            borderRadius: BorderRadius.circular(10),

                            child: Image.memory(

                              \_selectedImage!,

                              fit: BoxFit.cover,

                            ),

                          ),

                        )

                      : Container(

                          width: 150,

                          height: 150,

                          decoration: BoxDecoration(

                            border: Border.all(color: Colors.grey),

                            borderRadius: BorderRadius.circular(10),

                          ),

                          child: const Center(

                            child: Text('Select Image'),

                          ),

                        ),

                ),

              ),

* **GestureDetector:** Detects taps on the image container to trigger \_pickImage.
* **\_selectedImage != null:** Checks if the user has already selected an image.
* If an image is selected, it shows the image.Otherwise, it shows a placeholder ("Select Image").

**Show Image (ProductListPage)**

 Future<void> \_deleteProduct(String id) async {

    try {

      await FirebaseFirestore.instance.collection('products').doc(id).delete();

    } catch (e) {

      print('Error deleting product: $e');

    }

  }

**FirebaseFirestore.instance.collection('products'):**

Accesses the Firestore database and the products collection.

**.doc(id):**

Targets a specific document in the products collection using the provided id.

**.delete():**

Deletes the specified document from Firestore.

  floatingActionButton: FloatingActionButton(

        onPressed: () {

          Navigator.push(

            context,

            MaterialPageRoute(builder: (context) => const AddProductPage()),

          );

        },

        child: const Icon(Icons.add),

      )

floating action button (FAB) to the app, which is commonly used for primary actions like adding a new item.

**Navigator.push:**

Pushes a new route (screen) onto the navigation stack.

Moves the user to a new page while keeping the current page in memory.

**MaterialPageRoute:**

Creates a new route (screen) with a transition animation.

The builder parameter specifies the widget to display (AddProductPage).

**context:**

Provides information about the current location in the widget tree, required by Navigator to manage navigation.

**const AddProductPage():**

Creates an instance of the AddProductPage widget.

body: StreamBuilder<QuerySnapshot>(

        stream: FirebaseFirestore.instance.collection('products').snapshots(),

        builder: (context, snapshot) {

          if (snapshot.hasError) {

            return const Center(child: Text('Error loading products'));

          }

          if (snapshot.connectionState == ConnectionState.waiting) {

            return const Center(child: CircularProgressIndicator());

          }

          final products = snapshot.data!.docs;

          return GridView.builder(

            gridDelegate: const SliverGridDelegateWithFixedCrossAxisCount(

              crossAxisCount: 2,

              childAspectRatio: 0.8,

            ),

            itemCount: products.length,

            itemBuilder: (context, index) {

              final data = products[index].data() as Map<String, dynamic>;

              final imageBase64 = data['image'] as String?;

              Uint8List? imageBytes = imageBase64 != null ? base64Decode(imageBase64) : null;

              return Card(

                child: Column(

                  children: [

                    if (imageBytes != null)

                      Image.memory(imageBytes, height: 100, fit: BoxFit.cover)

                    else

                      const Icon(Icons.image, size: 100),

                    Text(data['name'] ?? 'No Name'),

                    Text("\$${data['price']}"),

                    Text("Qty: ${data['quantity']}"),

                    Row(

                      mainAxisAlignment: MainAxisAlignment.spaceAround,

                      children: [

                        IconButton(

                          icon: const Icon(Icons.edit),

                          onPressed: () {

                            Navigator.push(

                              context,

                              MaterialPageRoute(

                                builder: (context) =>

                                    UpdateProductPage(productId: products[index].id, productData: data),

                              ),

                            );

                          },

                        ),

                        IconButton(

                          icon: const Icon(Icons.delete),

                          onPressed: () => \_deleteProduct(products[index].id),

                        ),

                      ],

                    ),

                  ],

                ),

              );

            },

          );

        },

Displays a list of products from a Firestore database in a grid layout using a StreamBuilder. Each product is displayed as a card with its image, name, price, and quantity, along with edit and delete buttons.

**StreamBuilder<QuerySnapshot>** is a widget in Flutter used to build a UI that updates in real-time as data changes. It listens to a stream of data and rebuilds the UI whenever the stream emits new data. In this case, it listens to changes in a Firestore collection.

**QuerySnapshot:**

* A snapshot of a query result from Firestore.
* Contains the documents (data) returned by the query.
* Firestore provides a stream of QuerySnapshot objects for real-time updates.
* Each snapshot represents the state of the collection at a specific point in time.

builder: (context, snapshot) {

          if (snapshot.hasError) {

            return const Center(child: Text('Error loading products'));

          }

          if (snapshot.connectionState == ConnectionState.waiting) {

            return const Center(child: CircularProgressIndicator());

          }

          final products = snapshot.data!.docs;

* If there's an error while fetching data, displays a message: 'Error loading products'.
* If the connection is still waiting for data, displays a loading spinner (CircularProgressIndicator).

**final products = snapshot.data!.docs;**

* Retrieves the list of documents (docs) from the Firestore snapshot.
* Each document represents a product.

**GridView.builder:**

Dynamically creates a grid of items.

**gridDelegate:**

Defines the grid's layout:

* crossAxisCount: 2: Two items per row.
* childAspectRatio: 0.8: Each item's height is 80% of its width.

**itemCount:**

Specifies the number of items to display (products.length).

**itemBuilder:**

A function to build each grid item (product card).

**final data = products[index].data() as Map<String, dynamic>;**

* **products:** A list of documents fetched from Firestore. Each document represents a product.
* **products[index**]: Accesses the document at the given index in the list.
* **products[index].data():** Retrieves the actual data (fields) of the document in the form of a Map.
* **as Map<String, dynamic>:** Casts the data to a specific type (Map<String, dynamic>), where:
* **String:** Represents the keys (e.g., 'name', 'price', 'image').
* **dynamic:** Represents the values (e.g., 'Product 1', 10.5, 'Base64EncodedString').

**final imageBase64 = data['image'] as String?;**

**data['image']:** Accesses the value associated with the 'image' key in the data map. In this case, it retrieves the Base64-encoded string of the image.

**as String?:** Casts the value to a nullable string (String?), meaning:

* If the 'image' key exists and its value is a string, imageBase64 will hold that string.
* If the 'image' key does not exist or its value is null, imageBase64 will be null.

**Uint8List? imageBytes = imageBase64 != null ? base64Decode(imageBase64) : null;**

**imageBase64 != null:** Checks if imageBase64 contains a value (i.e., it's not null).

**base64Decode(imageBase64):** Converts the Base64-encoded string into raw image bytes (Uint8List), which can be displayed as an image in Flutter.

* ? and :: A ternary operator that works like an if-else:
* If imageBase64 is not null, decode it using base64Decode.
* Otherwise, set imageBytes to null.

return Card(

                child: Column(

                  children: [

                    if (imageBytes != null)

                      Image.memory(imageBytes, height: 100, fit: BoxFit.cover)

                    else

                      const Icon(Icons.image, size: 100),

                    Text(data['name'] ?? 'No Name'),

                    Text("\$${data['price']}"),

                    Text("Qty: ${data['quantity']}"),

                    Row(

                      mainAxisAlignment: MainAxisAlignment.spaceAround,

                      children: [

                        IconButton(

                          icon: const Icon(Icons.edit),

                          onPressed: () {

                            Navigator.push(

                              context,

                              MaterialPageRoute(

                                builder: (context) =>

                                    UpdateProductPage(productId: products[index].id, productData: data),

                              ),

                            );

                          },

                        ),

                        IconButton(

                          icon: const Icon(Icons.delete),

                          onPressed: () => \_deleteProduct(products[index].id),

                        ),

                      ],

returns a Card widget that represents a single product in a grid or list. It displays the product's image, name, price, quantity, and two action buttons (edit and delete).

**if (imageBytes != null)**

**Condition:**

Checks if the product has an image (imageBytes is not null).

**If true:**

Displays the image using Image.memory(imageBytes), which renders the raw image data (Uint8List) from the Firestore document.

**If false:**

Displays a placeholder icon (Icons.image) with a size of 100 pixels.

**Text(data['name'] ?? 'No Name')**

* Displays the product name.
* If the name is missing (null), it defaults to 'No Name'.

**Text("\$${data['price']}")**

Displays the product's price with a dollar sign.

**Example:** If data['price'] is 10.5, it will display $10.5.

**Text("Qty: ${data['quantity']}")**

Displays the product's quantity.

**Example:** If data['quantity'] is 5, it will display Qty: 5.

**onPressed:**

Navigates to the UpdateProductPage when the button is clicked.

**Navigator.push:** Opens a new screen.

**UpdateProductPage constructor:** Passes the productId (Firestore document ID) and the productData (product details) to the update page.

**UpdateProductPage:**

class UpdateProductPage extends StatefulWidget {

  final String productId;

  final Map<String, dynamic> productData;

  const UpdateProductPage({Key? key, required this.productId, required this.productData}) : super(key: key);

  @override

  State<UpdateProductPage> createState() => \_UpdateProductPageState();

}

**productId (type: String):**

* This represents the ID of the product that we want to update.
* It is passed to this page so that we know which product to edit in the database.

**productData (type: Map<String, dynamic>):**

* This contains all the current details of the product (like name, price, quantity, and image).
* It is passed to this page to pre-fill the form fields with the current data of the product.

**required Keyword:**

* required makes sure that the parameters productId and productData must be provided when creating an instance of UpdateProductPage.
* It’s a way to enforce that the necessary data is available for the page to function properly.

**late TextEditingController \_nameController;**

The late keyword means that the \_nameController will be initialized later, but we promise to do so before it is used. It’s a way of telling Dart that the variable will definitely be initialized before being accessed.

void initState() {

    super.initState();

    \_nameController = TextEditingController(text: widget.productData['name']);

    \_priceController = TextEditingController(text: widget.productData['price'].toString());

    \_quantityController = TextEditingController(text: widget.productData['quantity'].toString());

    \_imageBase64 = widget.productData['image'];

  }

**initState()** is a special method in Flutter that is called once when the widget is inserted into the widget tree. It is used for initializing any state or variables before the widget is built for the first time.

This method is called automatically when the widget is created, and it's typically where you initialize values that depend on external data or need to be set up when the widget is created.

**super.initState();**

* This calls the initState() method of the parent class (State).
* It's important to call this because the parent class State has its own initialization logic that should be executed before any custom logic in the child class (\_UpdateProductPageState).
* This ensures the proper initialization of the widget lifecycle.

**\_nameController = TextEditingController(text: widget.productData['name']);**

* This initializes the \_nameController with the current product name from the passed productData.
* widget.productData['name'] accesses the name field from the productData map that was passed to the UpdateProductPage widget.

This will pre-fill the text field for the product name with the existing product name when the page is first shown. The user can then modify the name if needed.

-------------------------------------------xxxxxxxxxxxxxxxxxxxxxx--------------------------------------------

**Dynamic Dropdown in Flutter**

class \_ProvinceCityAreaDropdownPageState extends State<ProvinceCityAreaDropdownPage> {

  String? selectedProvince;

  String? selectedCity;

  String? selectedArea;

  List<String> provinces = [];

  List<String> cities = [];

  List<String> areas = [];

  final FirebaseFirestore \_firestore = FirebaseFirestore.instance;

* Declares three nullable string variables: selectedProvince, selectedCity, and selectedArea.
* These variables will hold the currently selected values for the province, city, and area dropdowns, respectively.
* The ? after String means the variables can hold null values (initially, nothing is selected).

Tracks the user's current selections for province, city, and area.

* Declares three lists of strings: provinces, cities, and areas.
* These lists will store the available options for the dropdown menus.
* Initially, the lists are empty and will be populated with data fetched from Firestore.

Holds the dropdown options for provinces, cities, and areas.

  @override

  void initState() {

    super.initState();

    \_loadProvinces();

  }

**@override:** Indicates this method overrides the default initState of State.

**void initState():** A lifecycle method called when the widget is inserted into the widget tree.

**super.initState();**Calls the parent class's initState method, ensuring any inherited initialization logic runs first.

Calls \_loadProvinces, which fetches the list of provinces from Firestore.if there is no province in databse all fields will be disabled ,then user add province so province dropdown will become enable and user can then select province to add city or area.

provinces = snapshot.docs.map((doc) => doc.id).toList();

Maps the document IDs (province names) to a list and assigns it to provinces.

**Snapshot**:Firestore query result containing documents.

**Docs**:List of documents from the Firestore query.

**.map**:Iterates over the list, applying a transformation function.

**Doc:**Represents a single document during iteration.

**((doc) => doc.id):**Defines a short inline function (lambda) that takes a single argument (doc) and returns its id.

void \_loadCities(String province) async {

    QuerySnapshot snapshot = await \_firestore.collection('provinces').doc(province).collection('cities').get();

    setState(() {

      cities = snapshot.docs.map((doc) => doc.id).toList();

      selectedCity = null;

      selectedArea = null;

      areas = [];

    });

  }

This function fetches all cities for a given province from Firestore and updates the dropdown menu and other related states. It ensures that when a new province is selected, the cities, selected city, and areas are reset accordingly.

**\_firestore.collection('provinces')** accesses the provinces collection in Firestore.

**.doc(province)** selects the specific document (province) using the province parameter.

**.collection('cities')** accesses the cities subcollection within the selected province document.

This line fetches all the cities under the selected province from Firestore and stores the result in snapshot.

**cities = snapshot.docs.map((doc) => doc.id).toList();**

* **snapshot.docs** retrieves all the documents (cities) fetched from Firestore.
* **.map((doc) => doc.id)** extracts the id (name of each city) from each document.
* **.toList()** converts the result into a list and assigns it to the cities variable.

Updates the cities list with the names of all cities fetched from Firestore.

**selectedCity = null;**

Resets the selectedCity variable to null.Clears the previously selected city because the province has changed, so the city dropdown needs to be reset.

**selectedArea = null;**

Resets the selectedArea variable to null.Clears the previously selected area since a new province (and therefore new cities) has been selected.

**areas = [];**

Clears the areas list by assigning an empty list to it.Resets the areas list because the province and city selection has changed, and new areas need to be fetched later.

QuerySnapshot snapshot = await \_firestore.collection('provinces').doc(province).collection('cities').doc(city).collection('areas').get();

**\_firestore.collection('provinces'):** Accesses the provinces collection in Firestore.

**.doc(province):** Selects the document corresponding to the given province.

**.collection('cities'):** Accesses the cities subcollection within the selected province document.

**.doc(city):** Selects the document corresponding to the given city.

**.collection('areas'):** Accesses the areas subcollection within the selected city document.

**.get():** Retrieves all documents (areas) in the areas subcollection and stores them in snapshot.

**Dropdown for Provinces:**

   DropdownButton<String>(

              hint: Text('Select Province'),

              value: selectedProvince,

              onChanged: (value) {

                setState(() {

                  selectedProvince = value;

                });

                if (value != null) \_loadCities(value);

              },

              items: provinces.map((province) {

                return DropdownMenuItem(value: province, child: Text(province));

              }).toList(),

            ),

**DropdownButton<String>:**

* Creates a dropdown menu that allows the user to select a province.
* <String> indicates that the dropdown values are strings.

**Value**: is a property of the DropdownButton widget in Flutter.

* It specifies the currently selected item in the dropdown menu.

Initially, **selectedProvince** is null, so the dropdown shows the placeholder (e.g., hint: Text('Select Province')).

* If the user selects a province (e.g., "Sindh"):
* The onChanged callback updates selectedProvince to "Sindh".
* The value property now equals "Sindh", so the dropdown displays it as the selected item.

**if (value != null) \_loadCities(value);**

**value:** This is the new value selected by the user from the dropdown. It comes from the onChanged callback of the DropdownButton.

**!= null:** Checks if the value is not null.

* If the user hasn't selected anything, value will be null, and this block will not execute.
* If the user selects a valid province, value will contain the name of the selected province (e.g., "Sindh").

**\_loadCities(value):**

* Calls the \_loadCities function and passes the selected province (value) as an argument.
* This function fetches the cities associated with the selected province from Firestore and updates the cities list dynamically.

**items: provinces.map((province)**

**items**: This is a property of the DropdownButton widget. It defines the list of items (options) that will appear in the dropdown menu.

**provinces:** A list of strings that contains the names of all available provinces (e.g., ["sindh", "Punjab", "KPK"]).

**.map((province) {...}):**

This is a Dart method that applies a function to each item in the provinces list.

The function takes each province (e.g., "Sindh") and converts it into a DropdownMenuItem widget.

**DropdownMenuItem:**

A Flutter widget that represents one selectable option in the dropdown menu.

The dropdown menu is made up of multiple DropdownMenuItem widgets.

**value: province**

* Assigns a unique value to this specific menu item.
* In this case, the value is set to province, which is the name of the province (e.g., "Sindh").
* When the user selects this item from the dropdown, the value of this item (e.g., "Sindh") is passed to the onChanged callback of the DropdownButton.To identify which item the user selected, so you can update the app's state accordingly.

**child: Text(province)**

* Specifies what the user will see for this menu item in the dropdown.
* Text(province) creates a text widget displaying the name of the province (e.g., "Sindh").
* The child property of the DropdownMenuItem is used to show the label or visual representation of the item in the dropdown menu.
* To provide a user-friendly way to display the options (e.g., showing "Sindh" as text for the user to select).

***Value: "Sindh" (used internally when the user selects it).***

***Child: Text("Sindh") (what the user sees in the dropdown).***

DropdownButton<String>(

              hint: Text('Select City'),

              value: selectedCity,

              onChanged: (value) {

                setState(() {

                  selectedCity = value;

                });

                if (value != null && selectedProvince != null) \_loadAreas(selectedProvince!, value);

              },

              items: cities.map((city) {

                return DropdownMenuItem(value: city, child: Text(city));

              }).toList(),

            ),

**onChanged: (value) { ... }**

* Defines what happens when the user selects a different city from the dropdown.
* The value parameter contains the name of the selected city (e.g., "Toronto").

**Inside the onChanged callback:**

**setState(() { selectedCity = value; });**

* Updates the selectedCity variable with the new value.
* Triggers a UI update to reflect the change.

**if (value != null && selectedProvince != null) \_loadAreas(selectedProvince!, value);**

* Checks if both a province and a city are selected.
* If true, calls \_loadAreas(selectedProvince!, value) to fetch the areas for the selected city and update the areas dropdown.

**items: cities.map((city) { ... }).toList(),**

* Generates the list of options for the dropdown menu from the cities list.

**Inside .map((city) { ... }):**

* Takes each city in the cities list (e.g., "Karachi", "Hyderabad") and creates a DropdownMenuItem for it.

**DropdownMenuItem(value: city, child: Text(city)):**

* **value**: city: Sets the value of the menu item to the name of the city.
* **child**: Text(city): Displays the city name as text in the dropdown

**Add ProvinceCityAreaPage**

class \_ProvinceCityAreaPageState extends State<ProvinceCityAreaPage> {

  final TextEditingController \_provinceController = TextEditingController();

  final TextEditingController \_cityController = TextEditingController();

  final TextEditingController \_areaController = TextEditingController();

  final FirebaseFirestore \_firestore = FirebaseFirestore.instance;

  String? selectedProvince;

  String? selectedCity;

  List<String> provinces = [];

  List<String> cities = [];

**\_provinceController, \_cityController, \_areaController:** Control input fields for adding provinces, cities, and areas.

**selectedProvince, selectedCity:** Hold the currently selected province and city.

**provinces, cities:** Store lists of provinces and cities retrieved from Firestore.

**\_firestore:** Provides access to the Firestore database.

**Initializing Provinces:**

Automatically loads the list of provinces when the page is created.

 @override

  void initState() {

    super.initState();

    \_loadProvinces();

**Adding a Province:**

void \_addProvince(BuildContext context) async {

    String province = \_provinceController.text.trim();

    if (province.isNotEmpty) {

      await \_firestore.collection('provinces').doc(province).set({'name': province});

      \_loadProvinces(); // Refresh provinces list.

      ScaffoldMessenger.of(context).showSnackBar(SnackBar(content: Text('Province Added Successfully')));

    } else {

      ScaffoldMessenger.of(context).showSnackBar(SnackBar(content: Text('Province name is required')));

    }

  }

**String province = \_provinceController.text.trim();**

Reads the text entered in the \_provinceController (linked to the province input field) and removes any leading or trailing whitespace using .trim().

**if (province.isNotEmpty)** {

Ensures that the user has entered a valid province name. If the name is empty, the function skips the database operations and shows an error message.

**await \_firestore.collection('provinces').doc(province).set({'name': province});**

Adds the new province to the Firestore database.

**\_firestore.collection('provinces'):** Accesses the provinces collection in Firestore.

**.doc(province):** Creates or accesses a document with the ID equal to the entered province name.

**.set({'name': province}):** Sets the name field of the document to the province name. If the document already exists, it will overwrite the data.

**\_loadProvinces();**

Calls the \_loadProvinces function to refresh the provinces list and update the UI with the newly added province.

**ScaffoldMessenger.of(context).showSnackBar(SnackBar(content: Text('Province Added Successfully')));**

Displays a success message using a SnackBar to inform the user that the province has been added.

**ScaffoldMessenger.of(context).showSnackBar(SnackBar(content: Text('Province name is required')));**

}

If the input is empty (province.isEmpty), this block runs to show an error message, reminding the user to enter a province name.

**Adding City:**

  void \_addCity(BuildContext context) async {

    if (selectedProvince == null) {

      ScaffoldMessenger.of(context).showSnackBar(SnackBar(content: Text('Please select a province')));

      return;

    }

    String city = \_cityController.text.trim();

    if (city.isNotEmpty) {

      DocumentReference provinceRef = \_firestore.collection('provinces').doc(selectedProvince);

      await provinceRef.collection('cities').doc(city).set({'name': city});

      \_loadCities(selectedProvince!); // Refresh cities list.

      ScaffoldMessenger.of(context).showSnackBar(SnackBar(content: Text('City Added Successfully')));

    } else {

      ScaffoldMessenger.of(context).showSnackBar(SnackBar(content: Text('City name is required')));

    }

  }

**selectedProvince == null:** Checks if no province is selected.

**ScaffoldMessenger.of(context).showSnackBar:** Displays a SnackBar message prompting the user to select a province.

**return;:** Stops further execution if no province is selected.

**DocumentReference provinceRef = \_firestore.collection('provinces').doc(selectedProvince);**

* Creates a reference to the Firestore document for the selected province.
* \_firestore.collection('provinces'): Accesses the provinces collection in Firestore.
* .doc(selectedProvince): Points to the document corresponding to the selected province.

**await provinceRef.collection('cities').doc(city).set({'name': city});**

* Adds the new city to the Firestore database under the selected province.
* **provinceRef.collection('cities'):** Accesses the cities sub-collection within the selected province document.
* **.doc(city):** Creates or accesses a document with the ID equal to the city name.
* **.set({'name': city}):** Sets the name field of the document to the city name. If the document already exists, it will overwrite the data.

**Add Area:**

Ensures that both a province and a city are selected before adding an area.

**selectedProvince == null || selectedCity == null:** Checks if either the province or city is unselected.

**ScaffoldMessenger.of(context).showSnackBar:** Displays an error message prompting the user to select both a province and a city.

**return;:** Stops further execution if the validation fails.

**String area = \_areaController.text.trim();**

Retrieves the text entered in the \_areaController (linked to the area input field) and removes any leading or trailing whitespace using .trim().

**if (area.isNotEmpty) {**

Ensures that the area name is not empty before proceeding to add it to Firestore.

**DocumentReference cityRef = \_firestore**

**.collection('provinces')**

**.doc(selectedProvince)**

**.collection('cities')**

**.doc(selectedCity);**

Creates a reference to the Firestore document for the selected city within the selected province.

**\_firestore.collection('provinces'):** Accesses the provinces collection in Firestore.

**.doc(selectedProvince):** Points to the document corresponding to the selected province.

**.collection('cities'):** Accesses the cities sub-collection within the selected province.

**.doc(selectedCity):** Points to the document corresponding to the selected city.

**await cityRef.collection('areas').doc(area).set({'name': area});**

Adds the new area to Firestore under the selected city.

**cityRef.collection('areas'):** Accesses the areas sub-collection within the selected city document.

**.doc(area):** Creates or accesses a document with the ID equal to the area name.

**.set({'name': area}):** Sets the name field of the document to the area name. If the document already exists, it will overwrite the data.