**1. Flutter UI Basics  
  
Flutter Basics Steps::**  
  
1. installation of Android studio.

2. make sure you have flutter folder sdk in your Pc.

3. add sdk flutter in environment variables.

4. run cmd: **flutter doctor**

5. **Opening in VSCode editor::**

i. open Vscode editor.

ii. drag a new folder with small letter(naming space) in vscode.  
iii. Run Ctrl+Shift+P

iv. write flutter and choose >**Flutter New Project**

v. choose (**Application**)

vi. Select a folder to setup your flutter project.

vii. then Open ­­

viii. name you application(**small**\_**letter**)

ix. press Enter

x. remove test(folder) and go to **main.dart** to start Coding

**Processing a startup of Flutter App**

* In Flutter, the flow starts with importing the Material package, which provides the tools and widgets for creating a modern app with Material Design.
* Next, the **main**() function is the entry point. It’s like turning on a machine, where we use **runApp**() to kickstart our app.
* From here, we define a **StatelessWidget**, which acts like a static blueprint—a structure that doesn’t change. It usually serves as the base or root of the app.
* Then, if we need something interactive or dynamic, we move to a **StatefulWidget**, which allows the app to change and adapt based on user actions or data updates. The State of this widget is where the magic happens—it handles and redraws the UI whenever updates occur.

**This flow ensures Flutter apps are modular, responsive, and easy to scale. Each step plays a critical role in building the app seamlessly.**

**1. Importing Material Library**

dart

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

* The flutter/material.dart package provides pre-designed widgets and tools for building a Material Design-based UI.
* This package is essential for creating apps with Flutter.

**2. main() Function**

dart

void main() {

runApp(const MainApp());

}

* **main()** is the entry point of every Flutter app.
* **runApp()** initializes the app and takes a widget as its argument (in this case, MainApp).
* The const keyword improves performance by marking MainApp as an immutable widget.

**3. MainApp Class**

dart

class MainApp extends StatelessWidget {

const MainApp({super.key});

@override

Widget build(BuildContext context) {

return Main();

}

}

* **StatelessWidget**:
  + This class is used when the widget doesn't maintain any state.
  + Once created, a StatelessWidget doesn't rebuild unless an external input changes.
* **MainApp Constructor**:
  + The const MainApp({super.key}) constructor ensures that the widget can be reused without being re-instantiated.
  + The super.key is a common practice for handling widget keys.
* **build() Method**:
  + The build method is required in every widget and is responsible for describing the widget tree.
  + It returns an instance of the Main widget.

**4. Main Class**

dart

class Main extends StatefulWidget {

const Main({super.key});

@override

State<Main> createState() => \_MainState();

}

* **StatefulWidget**:
  + Unlike StatelessWidget, a StatefulWidget can maintain internal state that changes dynamically.
  + The Main widget is declared as a stateful widget.
* **createState()**:
  + This method is used to create the state for the Main widget, which is \_MainState.

**5. \_MainState Class**

dart

class \_MainState extends State<Main> {

@override

Widget build(BuildContext context) {

return const Placeholder();

}

}

* **State**:
  + The \_MainState class holds the mutable state for the Main widget.
  + This class extends State<Main> to link with the Main widget.
* **build() Method**:
  + Similar to StatelessWidget, the build method in a StatefulWidget is used to describe the UI.
  + Here, it returns a Placeholder widget.
* **const Placeholder()**:
  + This widget is a simple visual indicator, used as a placeholder for the actual UI during development.

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Flutter UI is built using widgets, which are the building blocks of everything you see on the screen.

* **Widgets**: The foundation of Flutter UI; everything in Flutter is a widget, including buttons, text, and layouts. Widgets can be classified as:
  + **StatelessWidget**: These widgets don't change over time. They are ideal for static displays, like icons or static text.
  + **StatefulWidget**: These widgets change based on user interaction or data updates. Use these when you need to update UI in response to user actions or data changes.
* **Layouts**: Layouts in Flutter are created using widgets, which arrange other widgets on the screen. Common layout widgets include:
  + **Column** and **Row**: Arrange widgets vertically or horizontally.
  + **Container**: A versatile widget used for styling, positioning, and sizing.
  + **Stack**: Allows widgets to be placed on top of each other.
* **Theming and Styling**: Flutter provides customizable themes to give apps a consistent look and feel. You can set colors, fonts, and styling in a theme.

**2. State Management**

Managing the state, or data that changes over time, is central in Flutter. State management ensures the app’s UI updates in response to data changes.

* **setState**: Used with StatefulWidgets to update the UI. However, it’s only suited for simple apps.
* **Provider**: A popular package that lets you separate the UI and business logic. It uses the concept of *inherited widgets* to pass data down the widget tree efficiently.
* **Riverpod, BLoC, and MobX**: Advanced state management solutions, each with its unique approach to organizing and updating data.

**3. Data Handling and Storage**

For data management, Flutter supports both local and cloud storage solutions.

* **Local Storage**:
  + **SharedPreferences**: Stores small pieces of data, like user preferences, using a key-value format.
  + **SQLite**: Stores structured data locally in a database format, ideal for offline support in apps.
  + **Hive**: A NoSQL database optimized for Flutter, great for high-speed local data storage.
* **Remote Storage and APIs**: For dynamic data and real-time updates, connecting with back-end services is common.
  + **HTTP Requests**: Flutter’s http package allows you to send requests to APIs to fetch or submit data.
  + **Firebase**: A popular back-end service that provides real-time database, authentication, analytics, and storage solutions specifically designed for mobile apps.

**4. Back-End Connectivity**

Flutter integrates well with various back-ends, enabling functionality like user authentication, data retrieval, and more.

* **REST APIs**: Most back-end services provide REST APIs for interacting with data. In Flutter, you use packages like http to communicate with these APIs.
* **GraphQL**: An alternative to REST, GraphQL allows you to fetch only the data you need. Flutter has packages like graphql\_flutter for this purpose.
* **Firebase Integration**: Firebase offers a complete suite for Flutter, providing real-time database features, user authentication, storage, and more. It’s widely used due to its seamless integration with Flutter and minimal setup for developers.

**5. UI + Data + Back-End Integration**

Bringing it all together:

* **Provider with HTTP Requests**: Use a state management solution (like Provider) to manage the state of your data fetched from an API.
* **Data Models**: Structure the data from the API into data models in Flutter to make it easy to use and manage.
* **Widgets Connected to Data**: Use StatefulWidgets or Provider consumers to display and update UI elements based on data changes.

Flutter day2::  
\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_  
**Hour 1: Essential Widgets and Layouts**

Goal: Familiarize participants with layout and common widgets for UI design.

* **Widgets to Cover**:
  + Row, Column
  + ListView
  + Stack and Positioned
  + Expanded and Flexible
* **Hands-On**:
  + Build a simple UI: A profile page with an image, name, bio, and social media links using Column, Row, and Padding.

**Hour 2: Stateful Widgets and State Management**

Goal: Introduce stateful widgets and state management basics.

* **Topics to Cover**:
  + Difference between StatelessWidget and StatefulWidget
  + Managing state using setState
  + Handling user input with TextField, TextFormField, and validation
  + Buttons (ElevatedButton, TextButton, IconButton)
* **Hands-On**:
  + Create a form for user input (e.g., a login form) with validation.
  + Add a button to display entered data below the form using setState.

**Hour 3: Navigation and Routing**

Goal: Learn how to navigate between screens and pass data.

* **Topics to Cover**:
  + Navigator (push, pop)
  + Named routes
  + Passing arguments between screens
* **Hands-On**:
  + Build a mini app with two screens:
    1. Home screen: A button to navigate to a second screen.
    2. Details screen: Display data passed from the first screen.

**Hour 4: Lists and Dynamic Content**

Goal: Work with dynamic data in lists.

* **Topics to Cover**:
  + ListView.builder
  + Using Card and ListTile for lists
  + Adding icons and interactivity
* **Hands-On**:
  + Build a list of items (e.g., products or tasks).
  + Add functionality to delete an item when a button is clicked (e.g., using setState).

**Hour 5: Networking and Data Fetching**

Goal: Fetch data from a server using APIs.

* **Topics to Cover**:
  + Using http package to make API calls
  + Handling JSON data
  + Displaying fetched data in a ListView
* **Hands-On**:
  + Fetch and display a list of users or posts from a free API like https://jsonplaceholder.typicode.com/posts.
  + Add a loading indicator (CircularProgressIndicator) while data is being fetched.

**Hour 6: Mini Project**

Goal: Combine all concepts into a functional mini app.

* **Project Idea**: Build a To-Do App
  + **Features**:
    - Display a list of tasks (use ListView.builder).
    - Add a task using a form.
    - Mark tasks as completed with a checkbox.
    - Delete tasks.
    - Optionally, fetch initial tasks from an API and update the UI dynamically.
  + **Widgets/Concepts Used**:
    - ListView.builder, StatefulWidget, http, setState
    - Navigation (optional: for a separate "Add Task" screen)

**Additional Tips:**

* **Breaks**: Include 10-minute breaks between hours to keep participants fresh.
* **Hands-On First**: Always explain briefly and then dive into code.
* **Q&A**: Allocate 5-10 minutes at the end of each hour for questions.
* **Resources**: Share links to Flutter documentation and a GitHub repo with sample code after the session.

This structure ensures participants get practical knowledge and leave with a functional understanding of Flutter basics, including UI, state management, navigation, and data fetching. Let me know if you'd like more details on any section!