Lab 1 – Setting Up Flutter and Running Your First App

1. Objective

This lab helps students set up the Flutter environment and run their first Flutter application. Students will learn how to install tools, create a project, execute it on a device/emulator, and perform basic UI customization.

2. Requirements

Students must:

- Install Flutter SDK
- Configure IDE (Android Studio or VS Code)
- Create and run a Flutter app
- Customize the layout
- Submit screenshots as evidence

3. Guided Steps

Exercise 1 – Flutter Environment Setup

Goal: Install and verify Flutter tools.

Steps

1. Install Flutter SDK.
2. Add Flutter to PATH.
3. Run:

```
flutter doctor
```

4. Ensure all items are properly configured.
5. Verify Flutter & Dart plugins in Android Studio.

Deliverables

- Screenshot of flutter doctor
- Screenshot of Flutter/Dart plugin screen

Exercise 2 – Create and Run Your First Flutter App

Goal: Learn how to create a Flutter project and run the default app.

Steps

1. Open Android Studio → **New Flutter Project**
2. Use project name: hello_flutter_lab1
3. Explore key folders: lib/, android/, ios/, pubspec.yaml
4. Start emulator or connect device
5. Run the default counter app
6. Modify the AppBar title in main.dart:

title: 'My First Flutter App',

7. Use **Hot Reload** and observe changes

Deliverables

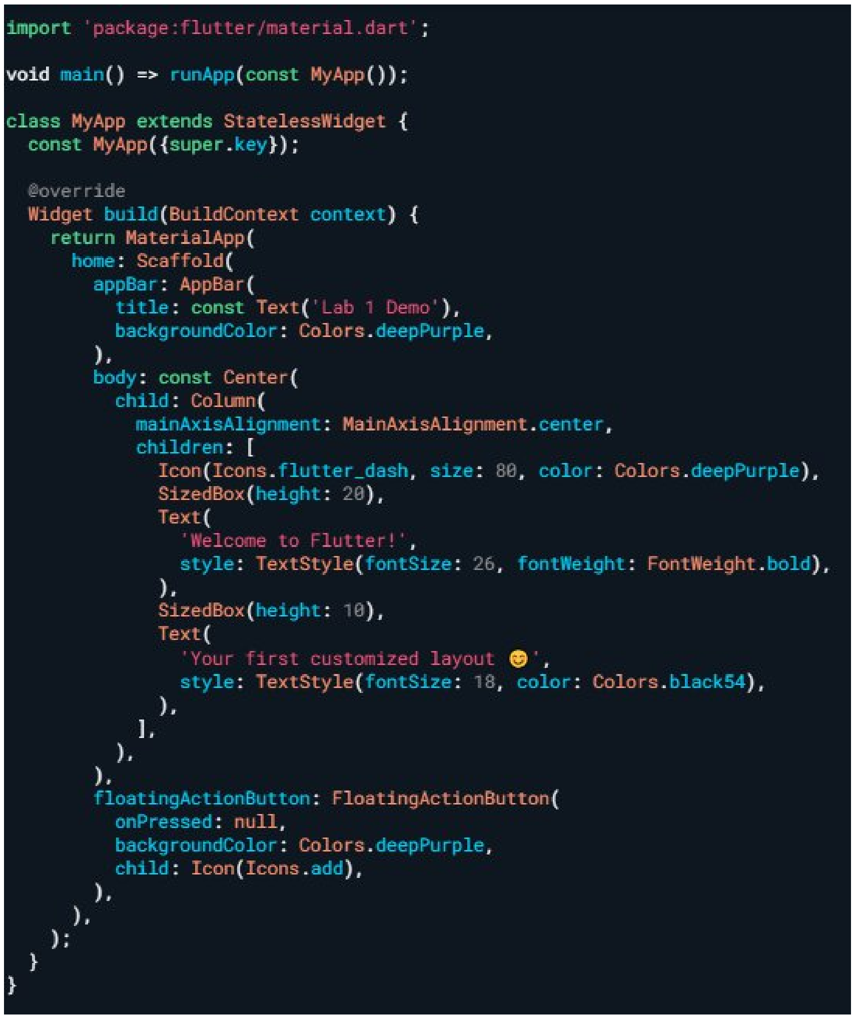
- Screenshot of project structure
- Screenshot of running counter app
- Screenshot showing updated title via Hot Reload

Exercise 3 – Customize Your First Flutter UI

Goal: Build a simple custom UI with widgets learned in class.

Steps

1. Replace default code in main.dart with:

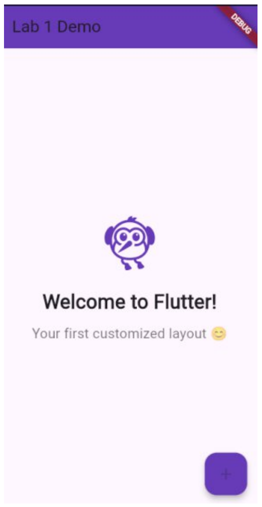


```dart
import 'package:flutter/material.dart';

void main() => runApp(const MyApp());

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

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Lab 1 Demo'),
          backgroundColor: Colors.deepPurple,
        ),
        body: const Center(
          child: Column(
            mainAxisAlignment: MainAxisAlignment.center,
            children: [
              Icon(Icons.flutter_dash, size: 80, color: Colors.deepPurple),
              SizedBox(height: 20),
              Text(
                'Welcome to Flutter!',
                style: TextStyle(fontSize: 26, fontWeight: FontWeight.bold),
              ),
              SizedBox(height: 10),
              Text(
                'Your first customized layout 😊',
                style: TextStyle(fontSize: 18, color: Colors.black54),
              ),
            ],
          ),
        ),
        floatingActionButton: FloatingActionButton(
          onPressed: null,
          backgroundColor: Colors.deepPurple,
          child: Icon(Icons.add),
        ),
      ),
    );
  }
}
```



2. Perform Hot Reload and verify UI updates.

3. Modify colors/text and reload again.

Exercise 4 – Reflection Questions

1. What is the purpose of the flutter doctor command?

2. What file acts as the entry point of a Flutter application?

3. Explain the difference between **Hot Reload** and **Hot Restart**.

4. How does runApp() build the widget tree?
5. Describe how Flutter’s architecture enables cross-platform development.

4. Expected Results

- Successful setup of Flutter environment
- Ability to create and run Flutter projects
- Understanding of widgets, layout, and hot reload
- Complete screenshots for all required steps

5. Submission

- Zipped Flutter project folder or GitHub link
- All screenshots
- Written answers for reflection questions

Lab 2 – Dart Essentials Practice Lab

1. Objective

This lab helps students build a strong foundation in Dart by practicing core language features including program structure, data types, variables, collections, control flow, functions, OOP basics, null safety, async/await, and streams.
All exercises can run directly on **DartPad**, **Android Studio**, or **VS Code**.

2. Requirements

Complete **five exercises** covering the fundamental concepts of Dart:

1. **Basic Syntax & Data Types** – practice printing output and using core types.
2. **Collections & Operators** – manipulate lists, sets, and maps using Dart operators.
3. **Control Flow & Functions** – implement logic using if/else, switch, loops, and functions.
4. **Intro OOP** – create classes, constructors, inheritance, and method overriding.
5. **Async & Null Safety** – work with async/await, Futures, and null-safety operators.

Each task must:

- Compile and run successfully.
- Print results to the console.
- Contain comments explaining the code.

3. Guided Steps

Exercise 1 – Basic Syntax & Data Types

Goal: Practice program structure and variable declarations.

Steps

1. Create a main() function.
2. Declare variables using: int, double, String, bool.
3. Use print() and **string interpolation** ($var, ${expr}) to show values.

Exercise 2 – Collections & Operators

Goal: Work with List, Set, Map and operators (+, -, ==, &&, ? :).

Steps

1. Create a List of integers.
2. Use arithmetic & comparison operators.
3. Create a Set (unique values) and a Map (key-value).
4. Use indexing, add(), remove(), and map access.

Exercise 3 – Control Flow & Functions

Goal: Apply if/else, switch, loops, and functions.

Steps

1. Write an if/else block to check score.
2. Write a switch case for day of week.
3. Loop through a collection using for, for-in, and forEach().
4. Create a function using normal and arrow syntax.

Exercise 4 – Intro to OOP

Goal: Practice classes, objects, constructors, inheritance, and overriding.

Steps

1. Create a class Car with one property and a method.

2. Create a named constructor.
3. Create a subclass ElectricCar that overrides a method.
4. Instantiate objects and print results.

Exercise 5 – Async, Future, Null Safety & Streams

Goal: Work with Dart's asynchronous features.

Steps

1. Create an async function using Future + await.
2. Use Future.delayed() to simulate loading.
3. Practice null-safety operators (?, ??, !).
4. Create a simple Stream of integers and listen to values.

4. Expected Results

- Students execute all exercises without errors.
- Output shows correct values, loop counts, OOP behavior, and async sequencing.
- Students understand Dart essentials needed for Module 3 onward.

5. Submission

- **Deliverable:** A single Dart file containing all exercises, or a zipped project.
- **Evaluation Criteria:**
  - Correctness (40%)
  - Use of Dart features (25%)
  - Output accuracy (20%)
  - Code readability & comments (15%)

Lab 3 – Advanced Dart Practice Exercises

1. Objective

This lab helps students strengthen their understanding of advanced Dart features—including asynchronous programming, streams, microtasks, JSON handling, and factory constructors—by completing five hands-on exercises that can run directly on DartPad or in Android Studio / VS Code.

2. Requirements

Complete **five mini-projects** demonstrating advanced Dart concepts:

1. **Product Model & Repository** – build a data model and a repository with both Future and Stream behaviors.
2. **User Repository with JSON** – simulate fetching and parsing JSON data from an API.
3. **Async + Microtask Debugging** – illustrate event-loop and microtask order.
4. **Stream Transformation** – apply functional operations such as map() and where() to a stream.
5. **Factory Constructors & Cache** – demonstrate singleton and factory patterns.

All tasks must:

- Execute without error in DartPad or Flutter's integrated Dart runtime.
- Include brief inline comments explaining logic and expected output.
- Print key results to the console.

3. Guided Steps

Exercise 1 – Product Model & Repository

Goal: Understand Futures and Streams.

1. Define Product { id, name, price }.
2. Implement ProductRepository with:

- Future<List<Product>> getAll()
- Stream<Product> liveAdded() for real-time updates.

3. Use StreamController.broadcast() to emit new items.
4. Print results to console.

Exercise 2 – User Repository with JSON

Goal: Practice JSON serialization / deserialization.

1. Create User { name, email } and User.fromJson(Map) constructor.
2. Simulate JSON list from an API.
3. Use Future<List<User>> to return parsed data.
4. Display results with print().

Exercise 3 – Async + Microtask Debugging

Goal: Differentiate microtask and event queues.

1. Write a snippet with scheduleMicrotask() and Future(() {...}).
2. Print execution order.
3. Explain why microtasks run before event callbacks.

Exercise 4 – Stream Transformation

Goal: Use functional stream operators.

1. Create a stream of numbers 1–5.
2. Transform values to their squares using map().
3. Filter even numbers with where().
4. Listen and print each emitted value.

Exercise 5 – Factory Constructors & Cache

Goal: Show how factory constructors implement caching.

1. Create a Settings class with private constructor.
2. Add a factory Settings() that returns a singleton instance.
3. Verify two instances refer to the same object (identical(a, b) → true).

4. Expected Results

- Each exercise compiles and prints its output in order.
- Console shows proper stream events, microtask execution sequence, and singleton confirmation.

- Students understand how to handle asynchronous flows and object creation patterns in Dart.

5. Submission

- **Deliverable:** Single Dart file or zipped project containing five completed exercises.
- **Evaluation Criteria:**
    - Correct implementation and syntax (40%)
    - Code clarity and comments (20%)
    - Output accuracy (20%)
    - Use of async / stream features (20%)

Lab 4 – Flutter UI Fundamentals

1. Objective

This lab helps students practice building user interfaces using Flutter's core widget system. Students will learn how to work with basic widgets, input controls, layout structures, navigation scaffolding, and design polish techniques such as spacing and theming.

This lab corresponds directly to **Module 4 – Flutter UI Fundamentals**.

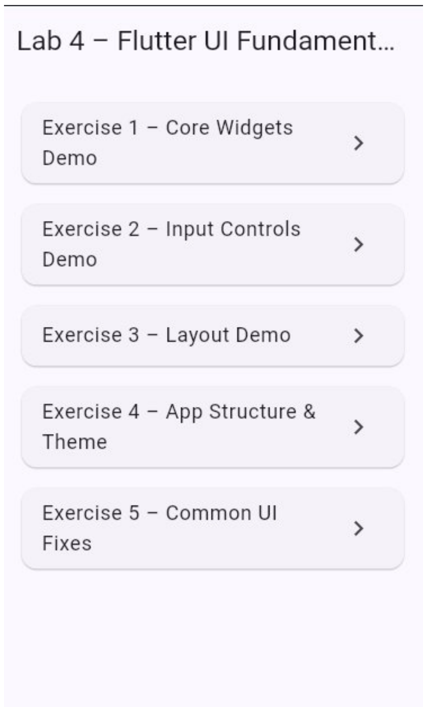
2. Requirements

Complete **five exercises** covering the main UI concepts:

1. Core Widgets (Text, Image, Icon, Card, ListTile)
2. Input Widgets (Slider, Switch, RadioListTile, Pickers)
3. Layout Composition (Column, Row, Padding, ListView)
4. Building Screen Structure using Scaffold
5. Applying ThemeData + fixing common UI errors

All tasks must:

- Compile and run in Android Studio / VS Code / DartPad
- Produce visible UI output
- Include comments explaining code
- Include screenshots (if required by instructor)


Lab 4 – Flutter UI Fundament...

Exercise 1 – Core Widgets Demo

Exercise 2 – Input Controls Demo

Exercise 3 – Layout Demo

Exercise 4 – App Structure & Theme

Exercise 5 – Common UI Fixes

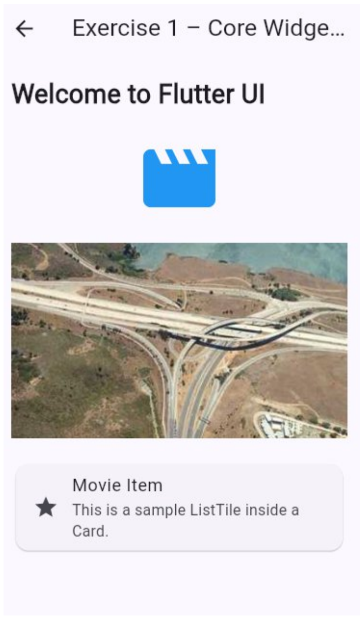
3. Guided Steps

Exercise 1 – Core Widgets: Text, Image, Icon, Card, ListTile

Goal: Build a simple UI demonstrating essential Flutter display widgets.

Steps

1. Create a new Flutter file core_widgets_demo.dart.
2. Build a simple screen that includes:
   - A headline Text
   - An Icon using Material Icons
   - An Image.network() (any valid image)
   - A Card containing a ListTile
3. Run the UI and confirm widgets display properly.



Welcome to Flutter UI

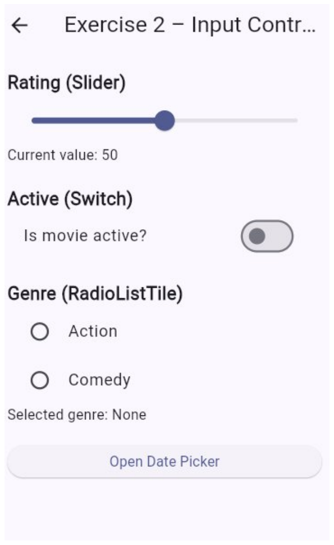
Movie Item
This is a sample ListTile inside a Card.

Exercise 2 – Input Widgets: Slider, Switch, RadioListTile, DatePicker

Goal: Build interactive UI that lets users control values.

Steps

1. Create a StatefulWidget named InputControlsDemo.
2. Implement a Slider, Switch, and RadioListTile group.
3. Add a button that shows a DatePicker when tapped.
4. Display updated values on screen.



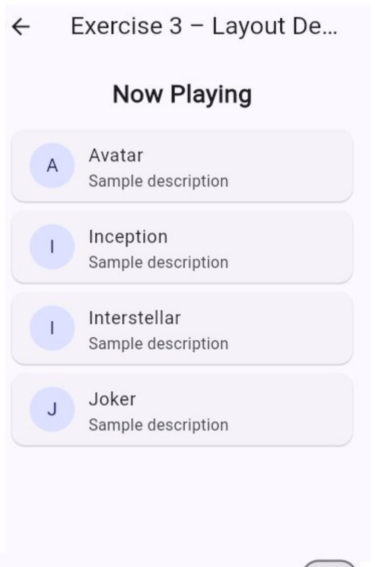
Rating (Slider)
Current value: 50
Active (Switch)
Is movie active?
Genre (RadioListTile)
Action
Comedy
Selected genre: None

Exercise 3 – Layout Basics: Column, Row, Padding, ListView

Goal: Build a sectioned UI layout similar to a real app Home screen.

Steps

1. Use Column to create vertical sections.
2. Add spacing using Padding and SizedBox.
3. Add a ListView.builder showing movie titles or items.
4. Apply consistent spacing (8, 12, or 16 px).


Exercise 3 – Layout De...

Now Playing

Avatar
Sample description

Inception
Sample description

Interstellar
Sample description

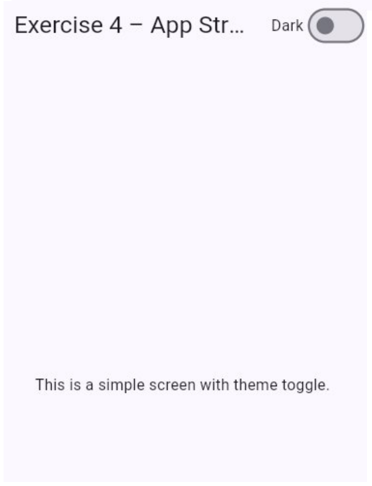
Joker
Sample description

Exercise 4 – App Structure with Scaffold, AppBar, FAB & Theme

Goal: Practice building a complete screen structure.

Steps

1. Create a new screen using Scaffold.
2. Add:
   - AppBar
   - Body
   - FloatingActionButton
   - Theme customization using ThemeData
3. Implement a "Dark Mode" toggle using themeMode.


Exercise 4 – App Str...

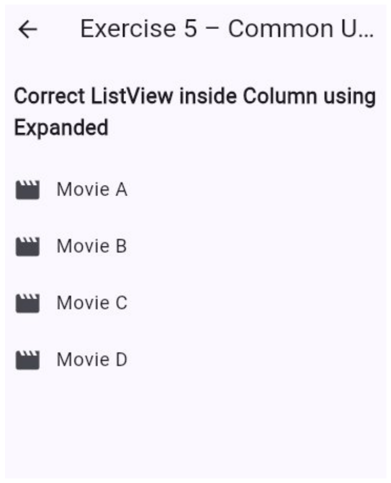
This is a simple screen with theme toggle.

Exercise 5 – Debug & Fix Common UI Errors

Goal: Understand common layout issues and fix them.

Tasks

1. Fix **ListView inside Column** using Expanded.
2. Fix **overflow** in small screens using SingleChildScrollView.
3. Fix state update issue by adding setState().
4. Fix DatePicker build context errors by calling from valid widget tree.


Exercise 5 – Common U...

Correct ListView inside Column using Expanded

Movie A

Movie B

Movie C

Movie D

4. Expected Results

By completing this lab, students should be able to:

- Build UI using core Flutter widgets
- Handle input widgets and pickers
- Compose complex layouts using Column/Row/ListView
- Structure screens using Scaffold & Theme
- Identify and fix common UI mistakes
- Produce visible and runnable Flutter UI screens

5. Submission

- Source code for all 5 exercises
- Screenshots of UI output for each
- Short explanation for fixes in Exercise 5
- Submit via LMS or upload zipped project

Lab 5 – Building A Movie Detail App With Navigation

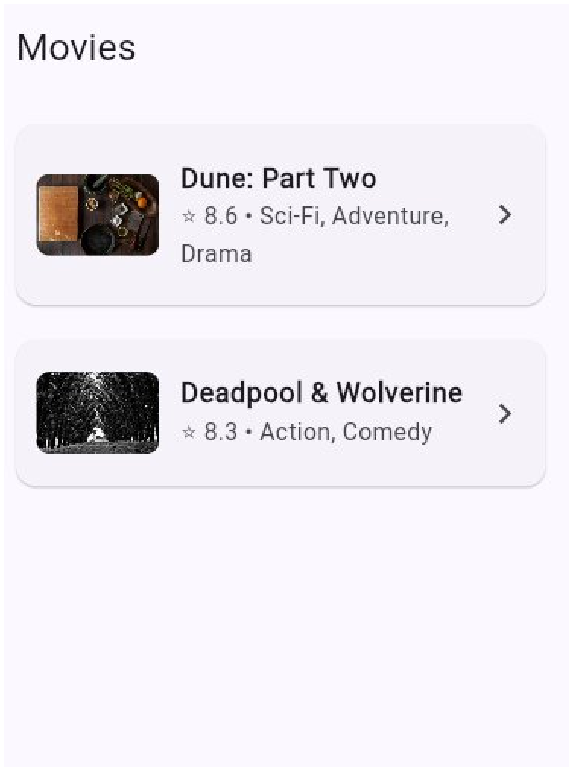
1. Objective

In this lab, students will design and implement a simple *Movie Detail App* that demonstrates how to use Flutter's navigation mechanisms (Navigator.push, MaterialPageRoute, or named routes) and basic UI composition.
The project reinforces the key ideas from **Module 5 – Navigation & State Management**.

2. Requirements

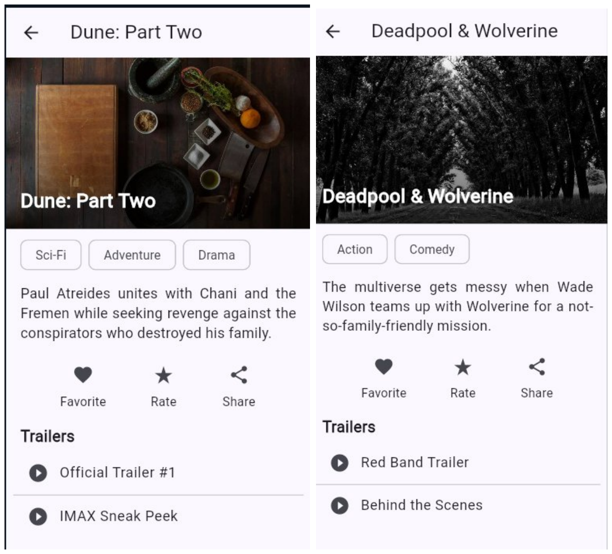
Create a two-screen Flutter application:

- **Home Screen** – displays a scrollable list of movies (poster, title, and rating).



- **Movie Detail Screen** – shows the selected movie's information, including:
    - Poster (Hero banner with gradient)
    - Title and genres displayed as chips
    - Overview text
    - Action buttons (Favorite / Rate / Share)

- List of trailers



Dune: Part Two

Dune: Part Two

Paul Atreides unites with Chani and the Fremen while seeking revenge against the conspirators who destroyed his family.

Trailers

Official Trailer #1

IMAX Sneak Peek

Deadpool & Wolverine

Deadpool & Wolverine

The multiverse gets messy when Wade Wilson teams up with Wolverine for a not-so-family-friendly mission.

Trailers

Red Band Trailer

Behind the Scenes

Technical Requirements

1. Navigation must use either
   - Navigator.push + MaterialPageRoute, or
   - Named routes (Navigator.pushNamed).
2. Pass a **Movie** object between screens.
3. Layout should be scrollable and responsive.
4. Use static sample data (no API calls).
5. The app must run correctly in **DartPad** and **Android Studio / VS Code**.

Optional Enhancements

- Implement a *Favorite* toggle that updates state.
- Add a simple search bar to filter movies.
- (Advanced) Experiment with a *deep-link* that opens a specific movie detail page.

3. Guided Steps

Step 1 – Project Setup
Create a new Flutter project (flutter create movie_app) or open DartPad Flutter.

Step 2 – Define Data Model
Create a Movie class with fields for id, title, posterUrl, overview, genres, rating, and a list of Trailers.
Add a sample_data.dart file containing 2–3 movie objects.

Step 3 – Build Home Screen
Use ListView.builder to display each movie card.
On tap → navigate to Movie Detail screen, passing the selected object.

Step 4 – Build Movie Detail Screen
Compose the UI progressively:

1. Blank Scaffold + AppBar
2. Hero Banner (Stack + Image.network + gradient)
3. Title & Genres (Column + Wrap + Chip)
4. Overview text with Padding
5. Row of IconButtons (Favorite, Rate, Share)
6. Trailer list using ListView.builder

Step 5 – Test and Polish
Run the app → check navigation works and UI scrolls smoothly.
Optionally add state management with setState() or ChangeNotifier.

4. Expected Result

When finished, the app should:

- Display a movie list on Home.
- Navigate to a detail page with poster, genres, overview, actions, and trailers.
- Return to Home using the Back button.
- Match the Demo Progress sequence in **Figures 8.12 – 8.18** of *Mastering Flutter 2025*.

5. Submission

- **Deliverable:** Zipped Flutter project or DartPad share link.

- **Evaluation Criteria:**
    - Correct navigation logic (30%)
    - UI layout & design consistency (30%)
    - Functionality & state handling (25%)
    - Code clarity & comments (15%)

Lab 6 – Building a Responsive Movie Genre Browsing Screen

1. Objective

In this lab, students will design and implement a **responsive movie browsing screen** that adapts to different screen sizes (small phones, large phones, tablets, and web).

The lab reinforces the key ideas from **Module 6 – Responsive UI & Adaptive Layouts**, including:

- Using **MediaQuery** to read screen dimensions
- Using **LayoutBuilder** to adapt to available space
- Using **Wrap**, **Expanded**, and **ListView/GridView** to build responsive layouts
- Building a real-world **Genre Filtering Screen** similar to the one shown in Figures 6.1–6.9 of *Mastering Flutter 2025*.

At the end, students will have a **fully functional responsive screen** that runs correctly in:

- DartPad (Flutter)
- Android Studio
- VS Code

2. Requirements

This lab is structured into three sub-tasks inside a single project:

- **Lab 6.1 – Responsive Hero & Heading Section**
- **Lab 6.2 – Search, Genre Chips & Sort Bar**
- **Lab 6.3 – Responsive Movie List & Tablet Layout**

You may complete these progressively in the same main.dart file.

2.1 Functional Requirements

Your responsive movie browsing screen must:

1. Display a title heading, e.g. **"Find a Movie"**.
2. Include a **search bar** that allows users to type a movie title or keyword.
3. Display a **set of genre chips** (Action, Drama, Comedy, etc.) that can be toggled on/off.
4. Include a **sort dropdown** with at least the following options:

    - A–Z
    - Z–A
    - Year
    - Rating

5. Show a **list of movie cards** for matching movies, including:
    - Poster image (use a simple network placeholder image)
    - Title
    - Year
6. Filter the visible list of movies based on:
    - Search text (case-insensitive match in title)
    - Selected genres (if any genre is selected, only show movies that have at least one of those genres)
7. Adapt the layout responsively:
    - On **small screens**: movie cards stacked in a **single column list**
    - On **wider screens (e.g. >= 800 px)**: movie cards arranged in **two columns** (e.g. using GridView.count)

2.2 Technical Requirements

- Use Flutter and Dart only (no third-party packages).
- The entire demo must run from a **single file** (e.g. main.dart), so it can be pasted into **DartPad → New Flutter**.
- Use **MediaQuery** and/or **LayoutBuilder** to implement breakpoints, not hard-coded “device names”.
- Use **SafeArea** to avoid notches / camera cutout issues.
- Use Wrap for genre chips so they automatically wrap to the next line.
- Code must compile and run without errors in:
    - DartPad (Flutter mode)
    - Android Studio

    - VS Code

2.3 Optional Enhancements (Bonus)

You may optionally:

- Add a small “badge” with the number of selected genres.
- Display movie rating as stars or numeric text.
- Add a “Clear filters” button.
- Provide different poster sizes for tablet vs phone using LayoutBuilder.

These are not required for passing the lab but can improve your score.

3. Guided Steps

Step 1 – Project Setup

1. Create a new Flutter project (e.g. flutter create lab6_responsive_ui)
   or open **DartPad → New Flutter**.
2. Replace the default main.dart content with the **Full Demo Code** at the end of this lab (or start from an empty MaterialApp and follow the steps).

Step 2 – Define the Movie Model & Sample Data

1. Create a simple Movie class with fields:
    - title (String)
    - year (int)
    - genres (List<String>)
    - posterUrl (String)
    - rating (double)
2. Define a constant list of sample movies (3–6 items) in a variable like allMovies.

This allows you to build UI first without backend/API.

Step 3 – Build the Base Scaffold

1. Create a ResponsiveMovieApp widget that returns a MaterialApp.

2. Set home to a GenreScreen (stateful widget).
3. Inside GenreScreen, wrap content in SafeArea and Padding.

The basic layout should look like:

- Title: “Find a Movie”
- Then vertical column for search bar, genres, sort, and movie list.

Step 4 – Implement a Responsive Search Bar

1. Add a TextField wrapped in a rounded container to simulate a search bar.
2. Add a state variable String searchQuery = '';.
3. In the onChanged callback of the TextField, update searchQuery using setState.
4. Use searchQuery later in your filtering logic to filter movies by title.

Step 5 – Implement Genre Chips Using Wrap

1. Declare a List<String> genres = [...] with some default genres.
2. Add a Set<String> selectedGenres = {}; or List<String> to track selected items.
3. Use a Wrap widget to display all genres as tappable chips.
4. When a chip is tapped:
    - If it was selected, remove it from selectedGenres.
    - Otherwise, add it.

The Wrap widget will automatically wrap chips to the next line, making it responsive.

Step 6 – Implement Sort Dropdown

1. Create an enum or string values to represent the sort options (A–Z, Z–A, Year, Rating).
2. Add a DropdownButton<String> bound to a state variable (e.g. selectedSort = "A-Z").
3. In the onChanged callback, update the sort variable and call setState.
4. Later, apply sorting to the filtered list before displaying it.

Step 7 – Filter and Sort the Movie List

1. Build a computed list (e.g. List<Movie> visibleMovies) inside build() based on:

    - Search query
    - Selected genres

2. Then sort visibleMovies according to selectedSort.
3. You can use .where() and .toLowerCase() for filtering, and .sort() for ordering.

Step 8 – Build a Responsive Movie List (ListView + LayoutBuilder)

1. Wrap the movie list area in an Expanded widget inside the main Column.
2. Use a LayoutBuilder to decide layout based on constraints.maxWidth:
    - If maxWidth < 800: use a ListView.builder with a single column.
    - Else: use a GridView.count with crossAxisCount: 2.
3. Each movie item should show:
    - Poster image (Image.network)
    - Title
    - Year

Use LayoutBuilder again inside a movie card to adjust poster size based on item width.

Step 9 – Test on Different Devices / Viewports

1. In DartPad:
    - Resize the preview pane
    - Switch orientation
2. In Android Studio / VS Code:
    - Run on a small phone emulator
    - Run on a larger phone
    - Run on a tablet emulator / Chrome Web

Confirm that:

- Chips wrap properly
- Movie list layout changes at the breakpoint

- Nothing overflows or gets clipped

4. Expected Result

When you finish the lab, your app should:

- Show a title “Find a Movie”.
- Display a search bar that filters the list by movie title (case-insensitive).
- Display a genre chip section at the top; selecting chips filters the list by genre.
- Provide a sort dropdown that affects the order of displayed movies.
- Adapt the layout based on screen width:
  - **Phones**: single-column vertical list of movie cards.
  - **Tablets / Wide web windows** (≥ 800 px): two-column movie layout.
- Match the look & feel of the Genre Screen from Figures 6.1–6.9 in *Mastering Flutter 2025* (not pixel-perfect, but functionally similar).

5. Submission

Deliverable:

- A zipped Flutter project *or* a **DartPad Flutter share link** containing the completed responsive screen.

Evaluation Criteria:

- **Responsive Layout Logic (30%)**
  Correct use of MediaQuery / LayoutBuilder and breakpoints for phone vs tablet.
- **UI Composition & Design (30%)**
  Clear layout with readable text, chip design, and movie cards.
- **Filtering & Sorting Functionality (25%)**
  Search, genre selection, and sort options work as expected.
- **Code Quality & Clarity (15%)**
  Clean, readable code with meaningful names and comments.

Lab 7 – Building a Signup Form with Validation & Good UX

1. Objective

In this lab, students will design and implement a user registration (signup) form that collects user input, validates it, and provides clear feedback to the user.

The lab reinforces the key ideas from **Module 7 – Forms & Validation**, including:

- Using Form and TextFormField for structured input
- Using FormState and GlobalKey to validate and save data
- Applying built-in and custom validation rules
- Managing focus and keyboard behaviors on mobile devices
- Improving form UX with helpful error messages and button states
- (Optional) Adding asynchronous validation before creating an account

At the end, students will have a fully functional signup form that runs correctly in:

- DartPad (Flutter)
- Android Studio
- VS Code

2. Requirements

This lab is structured into **four sub-tasks** inside a single project:

- **Lab 7.1 – Basic Registration Form (Form + TextFormField + Submit)**
- **Lab 7.2 – Validation Rules & Password Strength**
- **Lab 7.3 – Focus & Keyboard Management**
- **Lab 7.4 – Optional Async Validation (Email check)**

You may complete these progressively in the **same main.dart file** (or keep separate branches/versions for each step).

2.1 Functional Requirements

Your signup screen must:

1. **Collect user information**
   - Full name
   - Email
   - Password
   - Confirm password
2. **Validate user input**
   - All fields are required
   - Email must have a valid format (@ and . at minimum)
   - Password must respect strength rules:
     - Minimum 8 characters
     - At least 1 digit
   - Confirm password must match password
3. **Provide UX feedback**
   - Show inline error messages under invalid fields
   - Disable or block submission when form is invalid
   - Show a success message (e.g. SnackBar or dialog) when the form is valid
4. **(Optional – Async) Check email availability**
   - Simulate an API check (using Future.delayed)
   - If the email looks “already taken” (e.g. starts with "taken"), show an error message
   - Disable the submit button while checking

2.2 Technical Requirements

- Use **Flutter** and **Dart** only (no third-party packages).
- The entire demo must run from a single file (e.g. main.dart), so it can be pasted into **DartPad → New Flutter**.

- Use Form, TextFormField, GlobalKey<FormState>, and the validator: pattern.
- Use autovalidateMode in at least one version of the form (AutovalidateMode.onUserInteraction is recommended).
- Use FocusNode and FocusScope to move focus between fields and to dismiss the keyboard.
- Wrap the main form content in ListView or SingleChildScrollView to avoid overflow when the keyboard is open.
- Code must compile and run **without errors** in:
    - DartPad (Flutter mode)
    - Android Studio
    - VS Code

2.3 Optional Enhancements (Bonus)

You may optionally:

- Add a **Terms & Conditions** checkbox that must be accepted before submission.
- Add a **Show/Hide password** icon button.
- Add simple **password strength indicator** (e.g. Weak / Medium / Strong).
- Add a **loading indicator** in the submit button when performing async validation.
- Add extra fields such as username, phone, or role.

These are not required for passing the lab but can improve your score.

3. Guided Steps

Step 1 – Project Setup

1. Create a new Flutter project (e.g. flutter create lab7_forms_validation)
   or open **DartPad → New Flutter**.
2. Replace the default main.dart content with a minimal Flutter app that uses MaterialApp and a SignupScreen.
3. Verify that the app runs with a simple “Hello” text before adding any form logic.

Step 2 – Create the Basic Form Layout (Lab 7.1)

1. Inside SignupScreen, create a GlobalKey<FormState> named formKey.
2. Wrap your content with a Form widget and assign key: formKey.
3. Build the base layout:
    - Scaffold → AppBar(title: Text('Signup'))
    - body → Padding → Form → ListView (or SingleChildScrollView + Column)
4. Add the following TextFormFields:
    - Full Name
    - Email
    - Password
    - Confirm Password
5. Add a Submit button (e.g. ElevatedButton) below the fields.

Goal: At this stage, the form may not have any validation logic – it just collects input.

Step 3 – Add Basic Validation (Required Fields & Email) (Lab 7.1 → 7.2)

1. For each TextFormField, add a validator: function:
    - If the value is null or empty: return a short error message (e.g. "Name is required").
2. For the email field:
    - Check if it contains at least @ and .
    - If invalid, return "Enter a valid email".
3. In the onPressed of the Submit button:
    - Call final isValid = formKey.currentState!.validate();
    - If isValid is false, **do nothing else**.
    - If isValid is true, call formKey.currentState!.save(); and show a SnackBar confirming that the form is valid.

Goal: The form must not submit when required fields are empty or email is invalid.

Step 4 – Implement Password Rules & Confirm Password (Lab 7.2)

1. Move validation logic into separate functions (e.g. String? validateEmail(String? value)), to keep code clean.
2. For the password field:
   - Ensure at least 8 characters.
   - Ensure at least one digit (you can use RegExp(r'[0-9]')).
3. For the confirm password field:
   - Compare with the password value stored in state (e.g. _password variable).
   - If they do not match, return "Passwords do not match".
4. Enable autovalidateMode: AutovalidateMode.onUserInteraction on the Form to give quicker feedback as the user types.

Goal: The form enforces stronger password rules and ensures confirmation matches.

Step 5 – Manage Focus & Keyboard (Lab 7.3)

1. Create a FocusNode for each input field (name, email, password, confirm).
2. Assign these FocusNodes to the corresponding TextFormFields.
3. In textInputAction:
   - Set TextInputAction.next for all fields except the last one.
   - Set TextInputAction.done for the last field.
4. Use onFieldSubmitted: to move focus:
   - From Name → Email
   - From Email → Password
   - From Password → Confirm
   - From Confirm → Call _submit() / validate

5. Wrap the entire screen in a GestureDetector and in onTap, call FocusScope.of(context).unfocus() to dismiss the keyboard when tapping outside the form.

6. Ensure the form is wrapped in ListView or SingleChildScrollView to avoid overflow when the keyboard is open.

Goal: Users can move through fields smoothly with the keyboard and dismiss the keyboard easily.

Step 6 – Add Async Email Check (Optional, Lab 7.4)

1. Add a boolean bool isCheckingEmail = false;.

2. In your submit method:

   - First, run local validation:
     if (!formKey.currentState!.validate()) return;

   - If valid, set isCheckingEmail = true and call setState.

3. Simulate a server call:

   - await Future.delayed(const Duration(seconds: 2));

   - As a fake rule, treat emails starting with "taken" as already used.

4. If the email is taken:

   - Show a SnackBar or dialog: "This email is already taken".

   - Do not proceed to success message.

5. Regardless of the result, set isCheckingEmail = false and update the button:

   - While isCheckingEmail is true, disable the submit button and show a small CircularProgressIndicator instead of the text.

Goal: Demonstrate how to combine synchronous form validation with an asynchronous check before final submission.

4. Expected Result

When you finish the lab, your app should:

- Display a signup form with Full Name, Email, Password, and Confirm Password fields.

- Prevent submission when fields are empty or invalid.

- Show helpful error messages under each invalid field.
- Enforce password strength and ensure both password fields match.
- Allow users to navigate between fields using the Next/Done actions on the keyboard.
- Dismiss the keyboard when the user taps outside the fields.
- (Optional) Perform a fake asynchronous email check and show a message if the email is “already taken”.
- Run correctly in:
    - DartPad (Flutter mode)
    - Android Studio
    - VS Code

5. Submission

Deliverables:

- A zipped Flutter project **or** a DartPad Flutter share link containing the completed signup form.
- main.dart (or equivalent entry file) clearly showing:
    - Form structure
    - Validators
    - Focus/keyboard management
    - (Optional) async validation logic
- 2–3 screenshots of the app:
    - One with valid data and success message.
    - One with visible validation errors.

Evaluation Criteria:

- **Form Structure & Flow (30%)**
  Correct use of Form, TextFormField, FormState, and submit flow.
- **Validation Logic (30%)**
  Required fields, email format, password rules, confirm password, and error messages.

- **UX & Interaction (25%)**
  Focus management, keyboard behavior, inline errors, button states, and overall usability.
- **Code Quality & Clarity (15%)**
  Clean, readable code with meaningful names, separated validators, and minimal duplication.

LAB 8 – Building an API-powered List Screen (Flutter + REST API)

1. Objective

In this lab, students will build a simple Flutter screen that retrieves data from a public REST API, parses JSON into model classes, and displays the data using ListView.

This lab reinforces the key concepts from **Module 8 – Networking & JSON**, including:

- Making network requests with the http package
- Decoding JSON responses
- Converting JSON into Dart model classes
- Displaying asynchronous data using FutureBuilder
- Showing loading and error states
- (Optional) Sending data via POST requests

At the end, students will have a fully functional API-powered list screen that runs correctly in:

- **DartPad (Flutter)**
- **Android Studio**
- **VS Code**

2. Requirements

This lab is divided into **four sub-tasks**, each focusing on one major skill:

- **Lab 8.1 – Simple GET Request**
- **Lab 8.2 – JSON → Model Conversion & ListView**
- **Lab 8.3 – Loading + Error Handling**
- **Lab 8.4 – Creating an ApiService (Service Layer Pattern)**
- (Optional) **POST request: Create a new item**

You may complete these tasks progressively in the same project.

2.1 Functional Requirements

The final API-powered screen must:

Fetch data from a public REST API

- Use a safe and stable endpoint (e.g. https://jsonplaceholder.typicode.com/posts)

Parse JSON properly

- Convert raw JSON list → List<Model>
- Use factory Model.fromJson(Map<String, dynamic> json)

Display API results

- Render items using ListView or ListView.builder
- Show at least 1–2 fields (e.g. title, ID)

Handle async UI states

- Loading indicator (CircularProgressIndicator)
- Friendly error messages (e.g. "Something went wrong")
- Retry button *(optional)*

Optional: POST

- Simple form to send a POST request
- Display success or error

2.2 Technical Requirements

Your project must:

- Use Flutter + Dart only (no third-party network libraries)
- Use the http package for API calls
- Implement model parsing with dart:convert
- Use FutureBuilder to bind async data to the UI
- Use a separate **ApiService** class for networking
- Run without errors on:
  - DartPad

    - Android Studio
    - VS Code

Required Widgets & Methods:

- FutureBuilder
- ListView.builder
- CircularProgressIndicator
- json.decode()
- factory Model.fromJson()
- Exception handling

2.3 Optional Enhancements (Bonus)

Students may optionally:

- Add pull-to-refresh
- Add a detail screen when tapping an item
- Add POST request to create a new item
- Improve UI with cards, icons, or thumbnails
- Add a retry button inside the error state

These are not required but improve the final score.

3. Guided Steps

Step 1 – Project Setup

- Create a new Flutter project
  (flutter create lab8_networking)
  or open DartPad → New Flutter
- Replace main.dart with a minimal MaterialApp
- Verify the app runs correctly before adding logic

Step 2 – Call a Public API (Lab 8.1)

- Choose a sample endpoint: https://jsonplaceholder.typicode.com/posts
- Create a function to fetch data using http.get
- Understand response format (JSON array)
- Print raw JSON for verification

Goal: retrieve API data successfully.

Step 3 – Convert JSON → Model (Lab 8.2)

- Inspect the JSON structure and extract fields
- Create a Dart model class with:
    - fields
    - constructor
    - fromJson() factory
- Convert JSON list → List<Model>

Goal: correctly parse API response into model objects.

Step 4 – Display Data Using FutureBuilder (Lab 8.2 & 8.4)

- Wrap the API call with FutureBuilder<List<Model>>
- Handle states:
    - waiting → loading indicator
    - hasError → show error
    - hasData → show ListView
- Build a clean UI with ListTile or custom widgets

Goal: bind async API data to UI.

Step 5 – Handle Loading + Error States (Lab 8.3)

At minimum:

- Loading spinner
- Text message for errors

(Optional):

- Retry button
- Timeout handling
- Friendly fallback UI

Goal: professional asynchronous UX.

Step 6 – Organize Networking Code into ApiService (Lab 8.4)

- Create a dedicated class ApiService
- Move all fetch logic into this class
- Inject http.Client for testing
- Keep UI clean and focused on rendering

Goal: separate concerns using the Service Layer pattern.

Step 7 – Optional POST Request

- Create a simple form for a new item (title/body)
- Send POST using http.post
- Show success or error
- Update UI or print created data

Goal: demonstrate full Read + Create workflow.

4. Expected Result

Your finished app should:

- Successfully fetch data from a REST API
- Convert JSON → Dart models
- Display the list in a scrollable widget
- Show loading state before data arrives
- Show meaningful error messages on failure
- Use a clean architecture with ApiService
- (Optional) Allow creating new items with POST

Runs in:

- DartPad
- Android Studio
- VS Code

5. Submission

Deliverables

Students must submit:

- A zipped Flutter project or
- A DartPad share link (Flutter mode)

Submission must include:

- main.dart
- ApiService class
- Model class
- Screens for list (and optional POST)
- Screenshots:
    - loading state
    - error state
    - list successfully loaded

Evaluation Criteria

Category	**Weight**	**Description**
API Integration	**30%**	Correct GET request, decoding, proper use of http
JSON Parsing	**25%**	Accurate model creation & mapping
Async UI Handling	**25%**	FutureBuilder, loading, error states
Architecture	**10%**	Clean ApiService, separation of concerns
UX Quality	**10%**	Readability, clean layout, friendly error messages

Lab 8B – Practical REST API Integration Turning API Data into Useful Apps

1. Lab Title & Overview

Overview:
In this lab, students will build a small but practical Flutter app that consumes a real-world public REST API and uses the data for a meaningful purpose, not just as a simple list.

Students can choose **one** of the following scenarios (or more, for extra credit):

1. **Weather Companion App** – Use a weather API to help users decide what to do or what to wear.
2. **Movie & TV Explorer** – Use a movie API to help users find something interesting to watch.
3. *(Optional)* **Currency Rate Helper** – Use an exchange rate API to support simple travel or shopping decisions.

The lab reinforces everything from Module 8:

- HTTP + REST
- GET requests
- JSON parsing → model classes
- FutureBuilder + ListView / UI
- Loading & error states
- Service Layer pattern

2. Learning Objectives

After completing this lab, students will be able to:

- Integrate a **real public REST API** into a Flutter app.
- Convert JSON responses into **Dart models** that are easy to use.
- Design a **purpose-driven UI** where API data helps users make decisions.
- Implement loading, empty, and error states with a **good UX**.
- Organize networking code with a **service layer** for better structure.

3. Scenarios (Choose at least 1)

3.1 Scenario A – Weather Companion App

Goal of the app (real-world use):
Help the user answer questions like:

- "Do I need an umbrella today?"
- "Is it a good day for outdoor activities?"
- "How hot/cold will it be this afternoon?"

Suggested Features:

- Let the user:
  - Search or choose a city/region
  - View **current temperature**, **weather description**, and maybe an **icon**
  - See short info like "Feels like…", "Wind speed…", or "Humidity…"
- Use weather information to show:
  - A simple recommendation:
    - "Take an umbrella" / "No umbrella needed"
    - "Too hot for outdoor sports" / "Nice weather for a walk"

API Examples (for documentation purposes):

- Open weather APIs (e.g., Open-Meteo, OpenWeatherMap, etc.)
  (Teacher can suggest 1–2 concrete endpoints in class.)

3.2 Scenario B – Movie & TV Explorer App

Goal of the app (real-world use):
Help the user quickly discover something to watch tonight.

Suggested Features:

- Fetch a list of:
  - Trending movies

    - Popular TV shows
    - Or top-rated content
- Let the user:
    - Scroll through a list of titles + poster + rating
    - Tap on an item to see more details (overview, year, rating)
- Optional:
    - Add a simple “Watch later” / “Favorites” marker
    - Filter by rating or genre (client-side filter)

API Examples:

- Movie DB-style APIs (e.g., TMDB)
  (Teacher may provide a shared API key or use a demo endpoint in class)

3.3 (Optional) Scenario C – Currency Rate Helper

Goal of the app (real-world use):
Support basic decisions like:

- “If I have X VND, how much is that in USD?”
- “Is this price high compared to another currency?”

Suggested Features:

- Fetch latest exchange rates for a base currency (e.g., USD or EUR)
- Let the user:
    - Input an amount
    - Choose a target currency
    - See converted value
- Optional:
    - Show a short hint:
        - “Good time to buy” / “Rate is higher than average” (simple heuristic)

API Examples:

- Free exchange rate APIs (e.g., exchangerate.host, Frankfurter, etc.)

4. Functional Requirements (for all scenarios)

Regardless of the scenario chosen, the app must:

1. **Call a real REST API**
    - Use http package
    - Handle network errors gracefully
2. **Parse JSON into model classes**
    - Use at least one Dart model class with fields + fromJson
    - Avoid using raw dynamic maps all over the UI
3. **Display data with a meaningful UI**
    - Use cards, tiles, or sections that clearly show important information
    - Not just a debug Text(response.body)
4. **Handle loading & error states**
    - Show a loading indicator while waiting for API
    - Show an error message and a **Retry** action if the call fails
    - Handle empty or unexpected responses
5. **Use a Service Layer**
    - Separate API calls into a dedicated service class
    - Screens/widgets should call service methods, not http directly

5. Technical Requirements

- Flutter project (can be reused from Lab 8 or created new)
- Use http package for all network calls
- Use Future / async/await for asynchronous operations

- Use FutureBuilder or equivalent pattern for binding API data to UI
- Use at least:
    - 1 model class
    - 1 service class
    - 1 main screen

(Optional but recommended):

- Detail screen
- Basic form (for search, filters, or user input)

6. Suggested Lab Steps (For Students)

Step 1 – Choose a Scenario

Pick **Weather**, **Movie**, or **Currency** (or another approved by the instructor).

Step 2 – Explore the API

- Read the API docs
- Identify:
    - Base URL
    - Endpoint
    - Required parameters
    - JSON response structure

Step 3 – Design the UI (Paper or Figma-level, simple)

- Decide:
    - What information you want to show
    - How it helps the user make a decision
    - Where loading / errors will appear

Step 4 – Define Model Classes

- Identify key fields from JSON (id, title, temp, rate, etc.)

- Create model structure on paper or in code (but lab document không cần code).

Step 5 – Plan the Service Layer

- Decide function names like:
    - fetchWeatherForCity
    - fetchTrendingMovies
    - fetchExchangeRates
- Clarify input → output:
    - Input: location / query / base currency
    - Output: List<Model> or a single Model

Step 6 – Integrate Into UI

- Use FutureBuilder to:
    - call the service
    - show loading/error/data
- Map model data → UI widgets (cards, list tiles, text, icons).

Step 7 – Add a "Purpose-driven element"

- For Weather: "recommendation text" (bring umbrella?, good for running?)
- For Movies: highlight top N choices, or mark favorites
- For Currency: show converted value + simple hint

7. Deliverables & Submission

Students must submit:

1. **App description (short):**
    - Which scenario they chose
    - What problem their app helps the user solve
2. **Screenshots:**
    - Loading state

    - Error state (e.g., when network off or invalid URL)
    - Successful data display screen
    - Any optional features (favorites, form, recommendation text, etc.)
3. **Source code (project or repository link)**
    - main.dart
    - model file(s)
    - service file(s)
    - main screen(s)

(Trong file lab, bạn chỉ mô tả yêu cầu; code sinh viên tự làm như các lab trước.)

8. Evaluation Criteria (for teacher)

You can reuse khung chấm của Lab 8, tinh chỉnh nhẹ:

Category	**Weight**	**Description**
API Integration	25%	Correct use of HTTP + endpoint, API actually called
JSON & Model Design	20%	Good mapping JSON → Dart models
UI & UX	20%	Data clearly presented, purpose-driven, not just raw text
Async Handling	20%	Loading, error, retry states, no crashes
Architecture	10%	Service layer, separation of concerns
Creativity	5%	How meaningful/interesting the scenario & decisions are

LAB 9 – Working With Local JSON Storage (Flutter + File Persistence)

1. Objective

In this lab, students will build three small Flutter tasks that teach how to store, read, and update data locally using JSON files.

This lab reinforces key concepts from **Module 9 – JSON & Local Storage**, including:

- Reading JSON from assets
- Writing JSON files to app storage
- Maintaining local persistence between app restarts
- Implementing simple CRUD operations on JSON data
- Designing UI that reflects local data changes

At the end, students will have a fully functional set of JSON-powered screens that run correctly in:

- DartPad (where applicable)
- Android Studio
- VS Code

2. Requirements

This lab is divided into **three sub-tasks**, each focusing on one major skill.

Lab 9.1 – Read JSON From Assets (Local Embedded File)

Lab 9.2 – Save & Load JSON From Device Storage

Lab 9.3 – JSON CRUD Mini Database (Search + Edit + Delete)

Students may complete them in **separate Flutter projects** OR in one combined project (recommended).

2.1 Functional Requirements

Lab 9.1 – Read JSON From Assets

The app must:

- Include a JSON file in the project /assets/
- Load JSON using rootBundle.loadString()
- Parse JSON into a Dart List
- Display data using ListView
- Show at least 2 fields per item

Lab 9.2 – Save & Load JSON Using Local Storage

The app must:

- Read JSON file from application documents directory
- Save updates back to the JSON file
- Persist data across app restarts
- Provide minimal UI:
    - Add item
    - Show list
    - Save button (or auto-save)

Lab 9.3 – Local JSON CRUD + Search

The app must include:

- A JSON file used as a local mini-database
- Ability to:
    - Add item
    - Edit item
    - Delete item
    - Search/filter items
- Automatically save JSON after every change
- Clean and functional list UI

2.2 Technical Requirements

Your projects must:

- Use Flutter + Dart only
- Use dart:convert for JSON encoding/decoding
- Use path_provider for accessing local storage (Lab 9.2 & 9.3)
- Use setState or State Management from module 7 (optional)
- Run without errors on:
    - DartPad (Lab 9.1 and parts of Lab 9.3)
    - Android Studio
    - VS Code

Required Widgets & Methods:

- ListView / ListView.builder
- FloatingActionButton
- TextField
- jsonDecode / jsonEncode
- File read/write (Lab 9.2 & 9.3)
- StatefulWidget

2.3 Optional Enhancements (Bonus)

Students may optionally:

- Add toast/snackbar messages after saving
- Add pull-to-refresh
- Add multiple JSON files
- Add confirm dialog before delete
- Auto-generate incremental IDs

- Create custom UI components

These are optional but improve your score.

3. Guided Steps

Lab 9.1 – Read JSON From Assets

Step 1 – Project Setup

Create new project and add JSON file inside assets/data/.

Step 2 – Register Assets in pubspec.yaml

Enable JSON file under flutter → assets.

Step 3 – Load JSON File

Use async loading (rootBundle.loadString) to read file content.

Step 4 – Decode JSON

Convert loaded string into a Dart List using JSON decode.

Step 5 – Display Data

Use ListView to show item fields.

Lab 9.2 – Save & Load JSON From Device Storage

Step 1 – Add path_provider

Add required dependency to pubspec.yaml.

Step 2 – Create a Storage Service

Implement helper methods to:

- Locate app document directory
- Read existing JSON file
- Create a new file if not found
- Write updated JSON back to storage

Step 3 – Load Data at Startup

Read file during initState and load items into a list.

Step 4 – UI for Adding Items

TextField + Add button → append items to list.

Step 5 – Save Button

Write list to JSON file and show confirmation message.

Lab 9.3 – JSON CRUD Mini Database

Step 1 – Local Data Structure

Prepare an in-memory list (loaded from JSON file).

Step 2 – Search Bar

Filter list by name or keyword.

Step 3 – Add / Edit / Delete

- Add: Create new item with unique ID
- Edit: Update fields of selected item
- Delete: Remove item after confirmation

Step 4 – Auto-Save

Re-save JSON file after each CRUD action.

Step 5 – Improved UI

Use ListTile or custom widgets to display item details.

4. Expected Result

Your final Lab 9 output should:

- Correctly load JSON from assets (Lab 9.1)
- Correctly save JSON to device storage and reload after restart (Lab 9.2)
- Provide full CRUD interaction with JSON "database" (Lab 9.3)
- Feature responsive UI that updates on changes

- Run on:
  - Android Studio
  - VS Code
  - DartPad (Lab 9.1 & partial Lab 9.3)

5. Submission

Students must submit:

Required Deliverables

- A ZIP file of project(s)
- Or DartPad links for applicable tasks
- JSON files used in each lab
- Screenshots:
  - JSON loaded into list (Lab 9.1)
  - Saving & reloading data (Lab 9.2)
  - CRUD + Search UI (Lab 9.3)

Evaluation Criteria

Category	**Weight**	**Description**
JSON Reading	25%	Correctly loads & parses assets JSON
JSON Writing	25%	Proper save/load functionality
Local Persistence	20%	Data persists after restart
CRUD Functionality	20%	Add, edit, delete, search
UI/UX Quality	10%	Clear layout, smooth interaction

LAB 10 – Authentication, Session Management & Notifications

1. Overview

This lab guides students through building authentication and user management features in Flutter step by step.

Each lab task is implemented as a **separate runnable Flutter project**, allowing students to verify each feature independently.
After completing all parts, students integrate all features into a final full project.

Local Notification is a **mandatory part** of this lab to ensure students achieve **LO7 – integrating notifications in mobile applications**.

2. Learning Outcomes

After completing this lab, students will be able to:

- Implement authentication flows using mock backend and real REST APIs
- Manage user sessions using local persistence
- Implement auto-login and logout mechanisms
- Integrate Firebase Authentication (Google Sign-In)
- Integrate local notifications after key user actions (LO7)

3. Prerequisites & Tools

- Android Studio
- Android Emulator or physical device (API 33+ recommended)
- Flutter SDK (stable channel)
- Internet connection (required for REST API & Firebase)

4. Required Packages & References

- DummyJSON Authentication API
  https://dummyjson.com/docs/auth
- SharedPreferences
  https://pub.dev/packages/shared_preferences
- Firebase Authentication
  https://firebase.google.com/docs/auth
- Firebase packages
  firebase_core, firebase_auth, google_sign_in
- Flutter Local Notifications
  https://pub.dev/packages/flutter_local_notifications

5. Submission Rules

5.1 Required Submissions

Students must submit **ALL** of the following:

Part projects (each runs independently):

- Lab10_1_MockLogin
- Lab10_2_RealApiLogin
- Lab10_3_AutoLogin_Logout
- Lab10_4_FirebaseGoogleSignIn
- Lab10_5_Notification

Final integrated project:

- Lab10_Full

5.2 Naming Convention

- Lab10_1_MockLogin
- Lab10_2_RealApiLogin
- Lab10_3_AutoLogin_Logout

- Lab10_4_FirebaseGoogleSignIn
- Lab10_5_Notification
- Lab10_Full

5.3 Submission Format

- Each project must be compressed as a .zip file
- Each submission should include a short README describing:
    - How to run the app
    - Test account (if any)

6. LAB TASKS

Lab 10.1 – Mock Login (Backend Simulation)

Goal

Build a login screen with validation and simulate backend authentication.

Description

Students create a login UI and simulate a backend authentication process using a mock service.
This lab focuses on UI structure, validation, and basic authentication flow.

Steps

1. Create project Lab10_1_MockLogin
2. Build Login UI (email, password, login button)
3. Validate input using Flutter Form
4. Simulate authentication with a delay and mock token
5. Navigate to Home screen on success

Expected Output

- Valid credentials navigate to Home

- Invalid credentials display an error message

Lab 10.2 – Real REST API Login

Goal

Authenticate users using a real REST API (DummyJSON).

Description

This lab replaces mock authentication with a real HTTP login request.
Students practice handling API responses, loading states, and errors.

Steps

1. Create project Lab10_2_RealApiLogin
2. Send POST login request to DummyJSON
3. Parse token from response
4. Handle loading and error states
5. Navigate to Home on success

Expected Output

- Login succeeds using real API
- Errors are handled gracefully

Lab 10.3 – Auto Login & Logout

Goal

Persist authentication state and restore login automatically.

Description

Students implement session persistence using SharedPreferences and manage login state across app restarts.

Steps

1. Create project Lab10_3_AutoLogin_Logout

2. Save token after login
3. Implement SplashScreen routing
4. Auto-login if token exists
5. Logout clears session and returns to Login

Expected Output

- App skips login when token exists
- Logout forces re-authentication

Lab 10.4 – Firebase Authentication (Google Sign-In)

Goal

Integrate Google Sign-In using Firebase Authentication.

Description

Students configure Firebase and implement Google Sign-In as an alternative authentication method.

Steps

1. Create project Lab10_4_FirebaseGoogleSignIn
2. Configure Firebase project and SHA keys
3. Implement Google Sign-In flow
4. Display user profile information
5. Logout from Firebase and Google

Expected Output

- Google Sign-In works correctly
- User profile is displayed
- Logout returns to Sign-In screen

Lab 10.5 – Local Notification (MANDATORY – LO7)

Goal

Integrate local notifications in a Flutter application.

Description

Students integrate local notifications to enhance user experience after important user actions.

Steps

1. Create project Lab10_5_Notification
2. Setup flutter_local_notifications
3. Request notification permission (Android 13+)
4. Trigger notification manually for demonstration
5. Verify notification is displayed by the system

In the integrated project, the same notification method must be called after a successful user action (e.g. login success).

Expected Output

- Notification appears on the device

Lab10_Full – Final Integrated Project

Goal

Integrate all authentication and notification features into one complete application.

Integration Order (Recommended)

1. Base project from Lab 10.1
2. Replace mock login with real API login (Lab 10.2)
3. Add session persistence and Splash routing (Lab 10.3)
4. Add Firebase Google Sign-In (Lab 10.4)
5. Add Local Notification (Lab 10.5)

Minimum Required Features

- SplashScreen routing
- Real API login

- Auto-login and logout
- Firebase Google Sign-In
- Local notification integration

7. Final Checklist

- All 5 part projects run independently
- Lab10_Full integrates all required features
- Local notification works correctly
- Correct naming and submission format

8. Score Distribution (Percentage)

- Lab 10.1 – Mock Login: 15%
- Lab 10.2 – Real REST API Login: 20%
- Lab 10.3 – Auto Login & Logout: 25%
- Lab 10.4 – Firebase Google Sign-In: 20%
- Lab 10.5 – Local Notification (LO7): 10%
- Lab10_Full – Integrated Project: 10%

Total: 100%

LAB 11 — TESTING & DEBUGGING IN FLUTTER (TASKLY APP)

Module 11 — Flutter Mobile Development
Total Tasks: 5 (4 exercises + 1 final consolidation)
Total Grade: 100%

SECTION A — Learning Objectives

After completing Lab 11, students will be able to:

- Write **unit tests** for pure Dart logic (Task model, TaskRepository).
- Write **widget tests** to validate UI behaviors in Flutter.
- Perform **navigation tests** to ensure screens transition correctly.
- Perform **integration testing** to validate end-to-end flows.
- Use **Flutter DevTools** to inspect widget trees, constraints, rebuilds, and performance.
- Combine multiple test types into one complete **Flutter test suite**.

SECTION B — LAB INSTRUCTIONS

LAB 11.1 — Unit Testing: Task Model & Repository

Description

Implement unit tests to validate logic in the Task model and TaskRepository.

Tasks

1. Create folder: test/unit/
2. Create file: task_model_test.dart
    - Test default completed value
    - Test toggle() switches true ↔ false
3. Create file: task_repository_test.dart
    - Test addTask()

   - Test deleteTask()
   - Test updateTask()
4. Use the **AAA pattern** (Arrange → Act → Assert).
5. Fix any logic errors discovered by tests.

Deliverables

- test/unit/task_model_test.dart
- test/unit/task_repository_test.dart

LAB 11.2 — Widget Testing: Task List UI

Description

Write widget tests to validate UI rendering and task addition in TaskListScreen.

Tasks

1. Create file:
   test/widget/task_list_widget_test.dart
2. Write tests for:
   - **Empty State:** "No tasks yet. Add one!"
   - **Add Task:** enter text → tap add → verify UI updates
   - **Multiple Tasks:** add two tasks → verify both visible
3. Use pumpWidget(), enterText(), tap(), pump().

Deliverables

- test/widget/task_list_widget_test.dart

LAB 11.3 — Navigation Testing: Task List → Task Detail

Description

Write tests to verify navigation to TaskDetailScreen.

Tasks

1. Create file:
   test/widget/task_navigation_test.dart
2. Seed repository with at least one task.
3. Pump TaskListScreen.
4. Tap seeded task → call pumpAndSettle().
5. Validate:
   - AppBar title: “Task Detail”
   - TextField key: detailTitleField

Deliverables

- test/widget/task_navigation_test.dart

LAB 11.4 — Integration Testing & DevTools Debugging

Description

Perform a full flow integration test and inspect UI/performance using DevTools.

Part A — Integration Test

Tasks

1. Create file:
   test/widget/task_integration_test.dart
2. Simulate full flow:
   - Add “Original title”
   - Tap task → open detail
   - Edit → “Updated title”
   - Save
   - Verify updated title appears in list

Deliverables

- test/widget/task_integration_test.dart

Part B — DevTools Debugging

Tasks

1. Run Taskly in debug mode
2. Open DevTools:
   - **Widget Inspector**
   - **Performance Timeline**
3. Capture screenshots:
   - Widget tree
   - Performance trace
4. Write a short report (½–1 page):
   - One potential layout issue
   - One potential performance issue
   - How tests or DevTools help detect them

Deliverables

- DevTools screenshots (2)
- Debugging report (PDF/DOC)

LAB 11.5 — Final Consolidation: Complete Test Suite

Description

Combine all tests into a complete structured test suite.

Tasks

1. Organize folders:

```
test/
  ├– unit/
```

├– widget/

└ integration/

2. Ensure presence of:
    - Unit tests
    - Widget tests
    - Navigation tests
    - Integration tests
3. Run:

flutter test

4. Fix all failing tests.
5. Write a **Test Summary Document**:
    - number of tests
    - types of tests
    - behaviors validated
    - known limitations

Deliverables

- Full test suite
- Test summary PDF

SECTION C — SUBMISSION REQUIREMENTS

Students must submit:

1. Source Code ZIP

Must include:

- /lib
- /test

- DevTools screenshots
- Reports

2. Required Screenshots

- Widget Inspector
- Performance Timeline

3. Reports

- Debugging Report (Lab 11.4)
- Test Suite Summary (Lab 11.5)

4. Naming Convention

Module11_Testing_<StudentID>_<FullName>.zip

SECTION D — GRADING RUBRIC (100%)

Component	**Weight**
LAB 11.1 – Unit Testing	**20%**
LAB 11.2 – Widget Testing	**20%**
LAB 11.3 – Navigation Testing	**20%**
LAB 11.4 – Integration Testing + DevTools Debugging	**30%**
LAB 11.5 – Full Test Suite Consolidation	**10%**
TOTAL	**100%**

LAB 12 — Performance Optimization & Deployment

--

Section A — Introduction

In this lab, students will apply the performance optimization techniques learned in **Module 12** to improve the **Taskly** application.
Students will analyze performance issues, reduce unnecessary widget rebuilds, optimize images and assets, analyze app size, and finally produce a **Release build** ready for distribution.

The lab consists of four exercises:

1. **Exercise 12.1 — Optimize List Rebuilds**
2. **Exercise 12.2 — Image & Asset Optimization**
3. **Exercise 12.3 — App Size Analysis**
4. **Exercise 12.4 — Final Optimization & Deployment (Comprehensive)**

--

⭑ Section B — Lab Instructions

--

◈ Exercise 12.1 — Optimize List Rebuilds in Taskly

Objective

Reduce unnecessary widget rebuilds when adding/toggling/deleting tasks.

Tasks

1. Open the **Taskly** project used in Modules 11–12 (or your copied Module 12 version).
2. Review the current implementation: TaskListScreen + inline ListTile.
3. Apply optimizations:
   - Extract TaskTile into a separate widget
   - Use Selector<TaskProvider, List<Task>> instead of watching the entire screen
   - Add const to static widgets

    - Assign keys properly using ValueKey(task.id)

Deliverables

- Before/After screenshots showing rebuild differences
- Updated code files: task_list_screen.dart and task_tile.dart

🔷 Exercise 12.2 — Image & Asset Optimization

Objective

Optimize images and assets to improve performance and reduce app size.

Tasks

1. Add a small test icon or image to Taskly (PNG recommended).
2. Perform optimizations:
    - Resize the image to an appropriate resolution (e.g., 128×128)
    - Pre-cache frequently used images
    - Remove unused assets from pubspec.yaml

Deliverables

- Before/After asset comparison
- Code snippet showing precacheImage()
- List of removed unused assets (if any)

🔷 Exercise 12.3 — App Size Analysis

Objective

Analyze the app’s size and identify sources of large file contributions.

Tasks

1. Run the size analysis command:
2. flutter build apk --analyze-size
3. Open and review the generated HTML size report.

4. Record:
    - Total APK size
    - Top 3 components taking the most space
    - Which assets or dependencies can be optimized or removed

Deliverables

- Screenshot of the size report
- A written list of optimization suggestions

◈ Exercise 12.4 — Final Optimization & Deployment (Comprehensive)

Objective

Finalize optimization and generate a Release build ready for deployment.

Tasks

1. Run the app in **Profile mode**:
2. flutter run --profile

Observe FPS, jank, rebuild frequency, and responsiveness.

3. Apply final optimizations:
    - Remove unnecessary print() or debug logs
    - Add const where applicable
    - Remove unused assets/dependencies
    - Run flutter clean
4. Build a **Release APK**:
5. flutter build apk --release

Or build an **AppBundle** (recommended for Google Play):

flutter build appbundle --release

6. Install and test the Release APK on a physical device or emulator.

Deliverables

- Release APK or AAB file
- Screenshot of Taskly running in Release mode
- Completed performance checklist
- Short summary: *"Why the app is now ready for deployment*"

--

⭑ Section C — Submission Format

--

Students must submit a .zip file containing:

1. Code Folder

lab12/

  /taskly/

  /lib/

  /assets/

2. Report (PDF or DOCX)

- Results for exercises 12.1 → 12.4
- Screenshots
- Explanations & observations

3. Release Artifact

- app-release.apk or app-release.aab

--

⭑ Rubric (100%)

--

Component	Description	Weight
Exercise 12.1 – List Optimization	Widget extraction, Selector use, reduced rebuilds	**25%**
Exercise 12.2 – Image Optimization	Resizing, precaching, asset cleanup	**15%**
Exercise 12.3 – App Size Analysis	Size report findings & optimization proposals	**20%**
Exercise 12.4 – Final Optimization & Deployment	Profile testing, final cleanup, Release build	**30%**
Report Quality & Submission Format	Clear documentation, correct file structure	**10%**

Total: 100%

COURSE: PRM393 – MOBILE APPLICATION DEVELOPMENT (FLUTTER)
PROJECT (30% OF COURSE GRADE)
This assignment should take an average student who is up-to-date with tutorial work approximately 5 weeks
Learning Outcomes: LO1,LO2,LO3,LO4,LO5,LO6,LO7

Plagiarism is presenting somebody else's work as your own. It includes copying information directly from the Web or books without referencing the material; submitting joint coursework as an individual effort; copying another student's coursework; stealing or buying coursework from someone else and submitting it as your own work. Suspected plagiarism will be investigated and if found to have occurred will be dealt with failure of the course.

All material copied or amended from any source (e.g. internet, books) must be referenced correctly according to the reference style you are using.

Submission Requirements

Students must submit:

- Source code (.zip) containing the entire Flutter project
- Technical Report (.pdf) following the structure below
- Lack of either source code or report → not allowed to do assignment demonstration.

DETAILED SPECIFICATION

Students must design and develop a complete **Flutter mobile application** based on a real business scenario.

Your report must include:

1. Team Introduction

A brief introduction about the project team (3–5 members), including:

- Full names of all members
- Roles and responsibilities
- Summary of each member's contribution

2. Case Study

A complete description of the business scenario selected for the project.
Students may choose any domain such as:

- Online sales systems
- E-commerce for independent shops
- Service operation management
- Booking & reservation systems
- Productivity management
- Customer service applications

3. Business Analysis / System Design

Describe in detail how you performed business analysis and designed the system. You need to demonstrate your understanding of the system architecture and how the application was deployed.

- All functions in your application should be described
- Database design should be clarified
- A detailed description of any new technologies you find out (not in school) to develop applications.
- List all functional and non-functional requirements of the system
- Describe the application architecture (Provider/Bloc/MVVM)
- Provide UI flow and database/API design

4. Development Requirements

Your Flutter application must meet the following requirements:

- UI Implementation
- State Management
- Local or Remote Database
- Deployment Requirement
    - Build a Release APK or AppBundle
    - Show proof of running in Release Mode
- Testing Requirement:
    - Students must implement at least 1 Unit Test and 1 Widget Test to validate business logic and UI behavior.

5. Demo of your mobile application:

- Thorough all functions and explanations.
- Students must demonstrate all core features, state management flow, data handling, testing outputs, and performance optimization results.

6. Conclusion and Discussion: the pros and cons of the application. What you've learned anything through the development of this application. In the future, if having more time, what would you do to improve it?

7. Contribution: Evaluate the contribution of each member during the project

Topic	Team Effort	Member 1	Member 2	Member ...
Case Study Analysis	100%	Ex: 40%	Ex: 30%	Ex: 30%
Business analysis	100%			
System design	100%			
Implementation	100%			
Documentation	100%			

Your implementation:

- All source code must be zipped and uploaded to EduNext.
- Code's comments are required

Your demonstration (15 minutes):

- You will be required to briefly demonstrate your system (slide should be prepared). Prepare to answer the lecturer's questions

Evaluation		
Task	**Score**	**Condition**
Case study	10%	A case study certainly coherent
Business analysis	15%	All functions is designed as standard and structured in accordance with the business rules
System design	15%	A design architecture, database design is expressed
Conclusion and discussion	10%	The personal opinions should be clarified. The knowledge learned should be highlighted.
Demonstration	50%	Programs comply with the proposed design. Operation with good quality.

PROJECT SAMPLE

Objective:

Develop a **Flutter mobile application** that allows customers to browse, view, and purchase items from an online store.

Database:

Students may choose:

- SQLite
- Firebase Firestore
- REST API backend

Main Functions:

(Each function = 10% score)

1. Design database/API structure
2. Login screen
3. Product list screen
4. Product detail screen
5. Shopping cart screen
6. Checkout/billing screen
7. Notifications screen
8. Map (store location) screen
9. Messaging/chat screen
10. Apply state management (Provider/Bloc)