Lab 1 - Setting Up the Development Environment and Running Your First Flutter App

**Course:** Mobile App Development

**Lab Duration:** 3 hours

**Lab Objective:** Set up the Flutter and Dart development environment, understand project structure, and run a basic Flutter app.

# Prerequisites

Before beginning the lab, ensure you have access to:

1. A computer (Windows, macOS, or Linux)
2. A stable internet connection
3. Administrator access to install software

# Tools Required

You will need the following tools installed:

1. Flutter SDK: The software development kit for Flutter.
2. Dart SDK: Programming language used for both front-end (Flutter) and back-end.
3. Android Studio or Visual Studio Code: Integrated Development Environment (IDE) for Flutter development.
4. Android SDK: Required to run Android emulators or connect to physical Android devices.

# Step-by-Step Instructions

## Step 1: Installing Flutter SDK

### Download Flutter:

Visit the official Flutter installation page and select your operating system (Windows, macOS, or Linux).

### Extract Flutter:

Extract the downloaded .zip file (for Windows/macOS) or use package managers (for Linux).

Place the extracted folder in a suitable directory, like C:\src on Windows.

### Add Flutter to Path:

Open your system's environment variables and add the flutter/bin directory to the PATH.

**Windows:**

Go to Control Panel → System and Security → System → Advanced System Settings → Environment Variables.

Under "System Variables," find the Path variable, and add a new entry pointing to the flutter\bin directory.

**macOS/Linux:**

Open a terminal and run:

export PATH="$PATH:`pwd`/flutter/bin"

### Run Flutter Doctor:

Open a terminal/command prompt and run the command:

flutter doctor

This will check if all dependencies are installed correctly. If anything is missing, Flutter Doctor will provide suggestions on how to resolve it.

## Step 2: Installing Android Studio or Visual Studio Code

### Install Android Studio:

Download Android Studio and follow the installation process.

Once installed, go to File > Settings > Plugins, and search for and install the Flutter and Dart plugins.

OR

Install Visual Studio Code:

Download VS Code and install it.

Open VS Code and navigate to the Extensions section.

Search for and install the Flutter and Dart extensions.

## Step 3: Setting Up Android SDK and Emulator

### Android SDK Setup:

In Android Studio, go to Preferences > Appearance & Behavior > System Settings > Android SDK.

Ensure that the Android SDK tools are installed, particularly the SDK Platform and SDK Tools.

### Create an Android Virtual Device (AVD):

Open the AVD Manager from Android Studio by going to Tools > AVD Manager.

Click Create Virtual Device, select a phone model, and follow the prompts to create the virtual device.

Launch the emulator from the AVD Manager.

## Step 4: Creating Your First Flutter App

### Create a Flutter Project:

Open your terminal or IDE and navigate to a suitable directory.

Run the following command to create a new Flutter project:

flutter create my\_first\_app

Once the project is created, navigate into the project directory:

cd my\_first\_app

### Explore Project Structure:

Open the project in Android Studio or VS Code.

Explore the important files:

lib/main.dart: The entry point of the application.

pubspec.yaml: Used to manage dependencies and project settings.

android, ios: Directories for platform-specific native code.

Take note of the main.dart file structure:

import 'package:flutter/material.dart';

void main() {

runApp(MyApp());

}

class MyApp extends StatelessWidget {

@override

Widget build(BuildContext context) {

return MaterialApp(

home: Scaffold(

appBar: AppBar(title: Text("Hello World")),

body: Center(child: Text("Hello World")),

),

);

}

}

### Step 5: Running the Flutter App

### Run on Emulator:

Ensure your emulator is running by launching the virtual device from AVD Manager.

In the terminal or IDE, run the following command to start the app:

flutter run

The app should launch on the emulator with "Hello World" displayed on the screen.

### Run on a Physical Device (Optional):

Connect your Android device via USB and enable Developer Mode and USB Debugging.

Run the following command to deploy the app on your device:

flutter devices

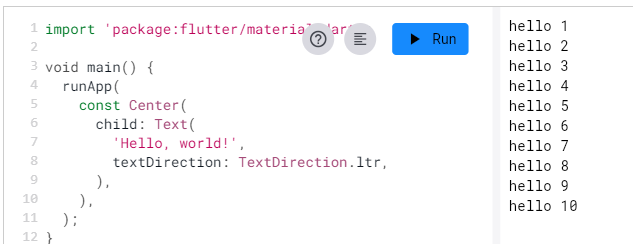
flutter run

## Building user interfaces with Flutter

## Flutter widgets are built using a modern framework that takes inspiration from React. The central idea is that you build your UI out of widgets. Widgets describe what their view should look like given their current configuration and state. When a widget's state changes, the widget rebuilds its description, which the framework diffs against the previous description in order to determine the minimal changes needed in the underlying render tree to transition from one state to the next.

## Hello world

## The minimal Flutter app simply calls the runApp() function with a widget:



The runApp() function takes the given Widget and makes it the root of the widget tree. In this example, the widget tree consists of two widgets, the Center widget and its child, the Text widget. The framework forces the root widget to cover the screen, which means the text "Hello, world" ends up centered on screen. The text direction needs to be specified in this instance; when the MaterialApp widget is used, this is taken care of for you, as demonstrated later.

When writing an app, you'll commonly author new widgets that are subclasses of either StatelessWidget or StatefulWidget, depending on whether your widget manages any state. A widget's main job is to implement a [build()](https://api.flutter.dev/flutter/widgets/StatelessWidget/build.html) function, which describes the widget in terms of other, lower-level widgets. The framework builds those widgets in turn until the process bottoms out in widgets that represent the underlying [RenderObject](https://api.flutter.dev/flutter/rendering/RenderObject-class.html), which computes and describes the geometry of the widget.

## Basic widgets

Flutter comes with a suite of powerful basic widgets, of which the following are commonly used:

Text

The Text widget lets you create a run of styled text within your application.

Row, Column

These flex widgets let you create flexible layouts in both the horizontal (Row) and vertical (Column) directions. The design of these objects is based on the web's flexbox layout model.

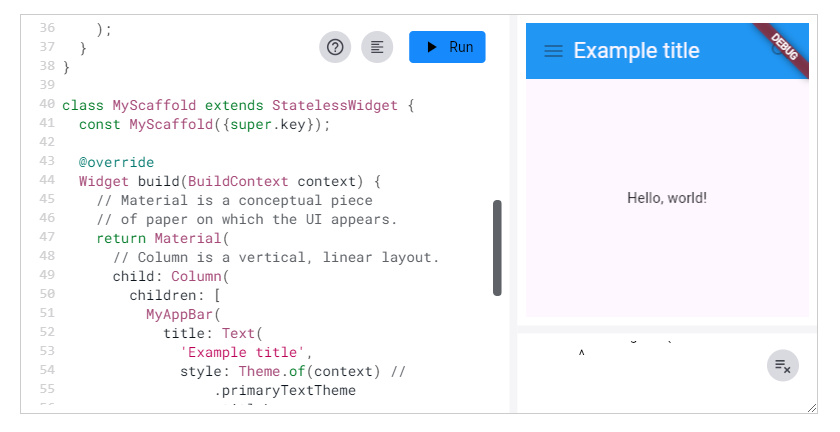
Stack

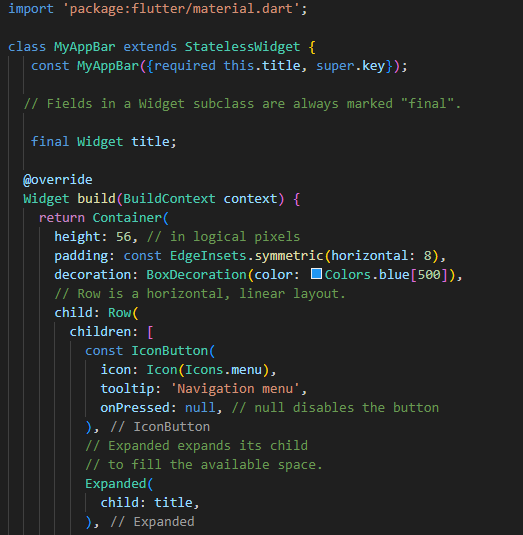
Instead of being linearly oriented (either horizontally or vertically), a Stack widget lets you place widgets on top of each other in paint order. You can then use the [Positioned](https://api.flutter.dev/flutter/widgets/Positioned-class.html) widget on children of a Stack to position them relative to the top, right, bottom, or left edge of the stack. Stacks are based on the web's absolute positioning layout model.

Container

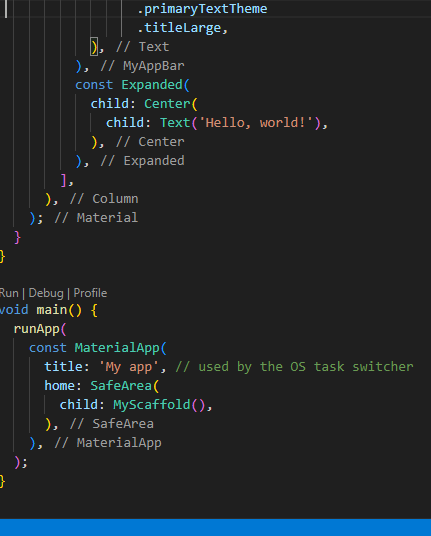
The Container widget lets you create a rectangular visual element. A container can be decorated with a BoxDecoration, such as a background, a border, or a shadow. A Container can also have margins, padding, and constraints applied to its size. In addition, a Container can be transformed in three-dimensional space using a matrix.

Below are some simple widgets that combine these and other widgets:

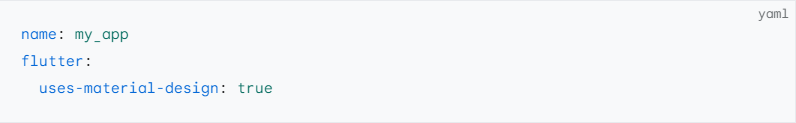








Be sure to have a uses-material-design: true entry in the flutter section of your pubspec.yaml file. It allows you to use the predefined set of Material icons. It's generally a good idea to include this line if you are using the Materials library.



Many Material Design widgets need to be inside of a MaterialApp to display properly, in order to inherit theme data. Therefore, run the application with a MaterialApp.

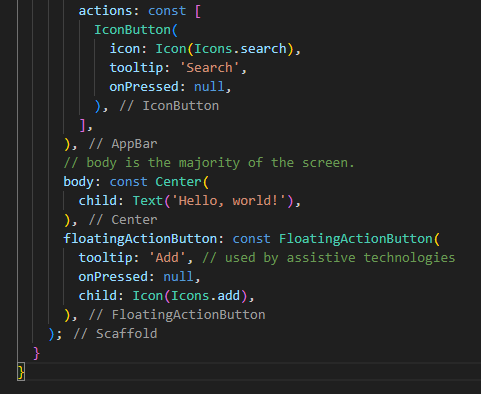
The MyAppBar widget creates a Container with a height of 56 device-independent pixels with an internal padding of 8 pixels, both on the left and the right. Inside the container, MyAppBar uses a Row layout to organize its children. The middle child, the title widget, is marked as Expanded, which means it expands to fill any remaining available space that hasn't been consumed by the other children. You can have multiple Expanded children and determine the ratio in which they consume the available space using the flex argument to Expanded.

The MyScaffold widget organizes its children in a vertical column. At the top of the column it places an instance of MyAppBar, passing the app bar a Text widget to use as its title. Passing widgets as arguments to other widgets is a powerful technique that lets you create generic widgets that can be reused in a wide variety of ways. Finally, MyScaffold uses an Expanded to fill the remaining space with its body, which consists of a centered message.

## Using Material Components

Flutter provides a number of widgets that help you build apps that follow Material Design. A Material app starts with the MaterialApp widget, which builds a number of useful widgets at the root of your app, including a Navigator, which manages a stack of widgets identified by strings, also known as "routes". The Navigator lets you transition smoothly between screens of your application. Using the MaterialApp widget is entirely optional but a good practice.



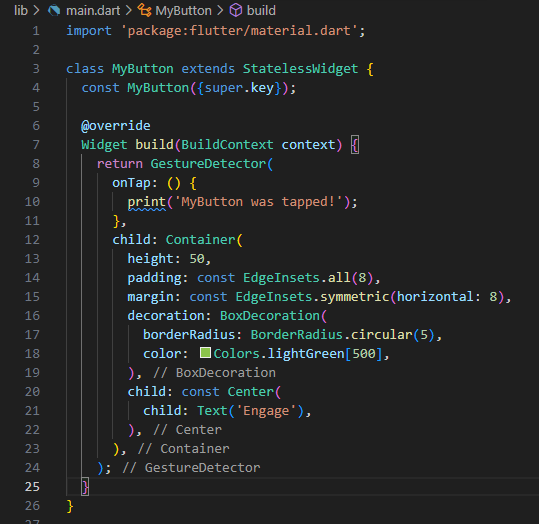


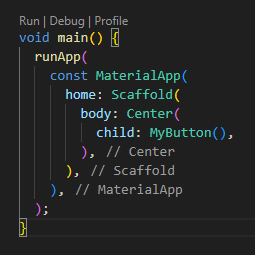
Now that the code has switched from MyAppBar and MyScaffold to the AppBar and Scaffold widgets, and from material.dart, the app is starting to look a bit more Material. For example, the app bar has a shadow and the title text inherits the correct styling automatically. A floating action button is also added.

Notice that widgets are passed as arguments to other widgets. The Scaffold widget takes a number of different widgets as named arguments, each of which are placed in the Scaffold layout in the appropriate place. Similarly, the AppBar widget lets you pass in widgets for the leading widget, and the actions of the title widget. This pattern recurs throughout the framework and is something you might consider when designing your own widgets.

## Handling gestures

Most applications include some form of user interaction with the system. The first step in building an interactive application is to detect input gestures. See how that works by creating a simple button:





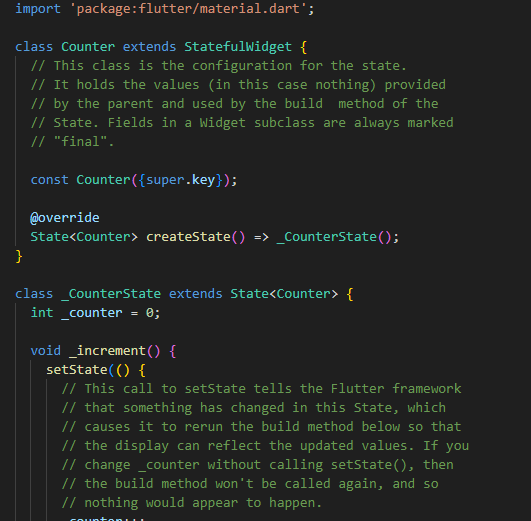
The GestureDetector widget doesn't have a visual representation but instead detects gestures made by the user. When the user taps the Container, the GestureDetector calls its onTap() callback, in this case printing a message to the console. You can use GestureDetector to detect a variety of input gestures, including taps, drags, and scales.

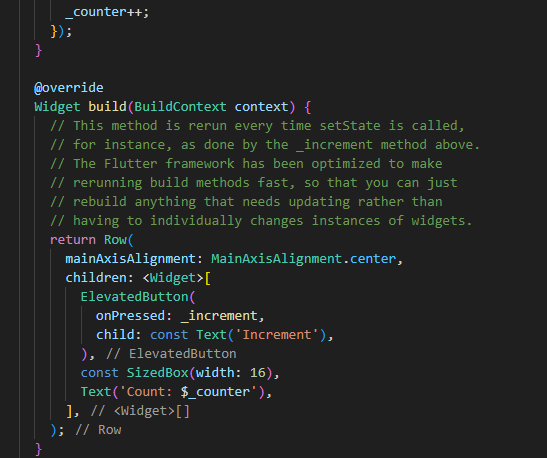
Many widgets use a GestureDetector to provide optional callbacks for other widgets. For example, the IconButton, ElevatedButton, and FloatingActionButton widgets have onPressed() callbacks that are triggered when the user taps the widget.

## Changing widgets in response to input

So far, this page has used only stateless widgets. Stateless widgets receive arguments from their parent widget, which they store in final member variables. When a widget is asked to build(), it uses these stored values to derive new arguments for the widgets it creates.

In order to build more complex experiences—for example, to react in more interesting ways to user input—applications typically carry some state. Flutter uses StatefulWidgets to capture this idea. StatefulWidgets are special widgets that know how to generate State objects, which are then used to hold state. Consider this basic example, using the ElevatedButton mentioned earlier:



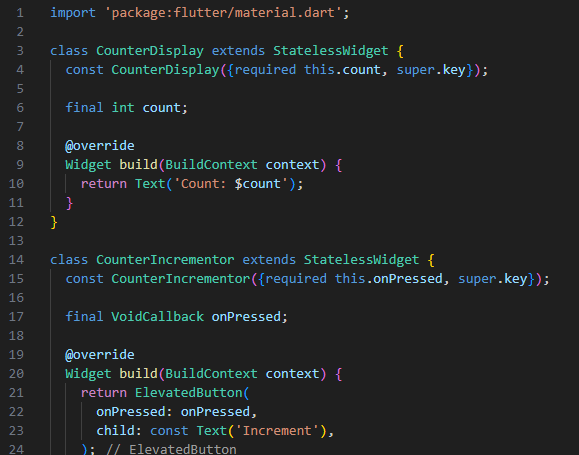


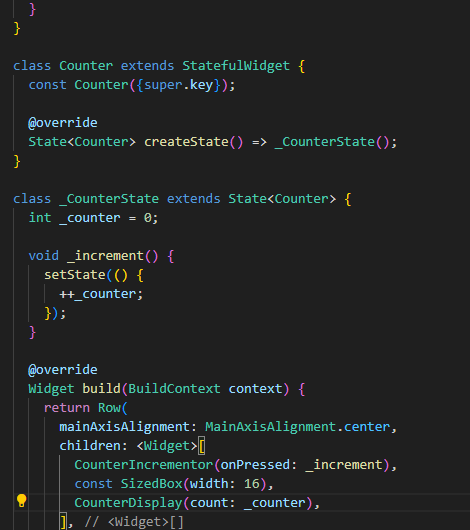


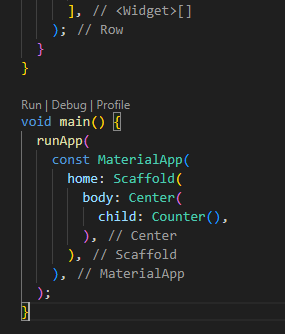
You might wonder why StatefulWidget and State are separate objects. In Flutter, these two types of objects have different life cycles. Widgets are temporary objects, used to construct a presentation of the application in its current state. State objects, on the other hand, are persistent between calls to build(), allowing them to remember information.

The example above accepts user input and directly uses the result in its build() method. In more complex applications, different parts of the widget hierarchy might be responsible for different concerns; for example, one widget might present a complex user interface with the goal of gathering specific information, such as a date or location, while another widget might use that information to change the overall presentation.

In Flutter, change notifications flow "up" the widget hierarchy by way of callbacks, while current state flows "down" to the stateless widgets that do presentation. The common parent that redirects this flow is the State. The following slightly more complex example shows how this works in practice:







Notice the creation of two new stateless widgets, cleanly separating the concerns of *displaying* the counter (CounterDisplay) and *changing* the counter (CounterIncrementor). Although the net result is the same as the previous example, the separation of responsibility allows greater complexity to be encapsulated in the individual widgets, while maintaining simplicity in the parent.

**Student Exercises:**

1. Set up the VS-Code with flutter and dart extension.
2. Implement the above code and design the first app.
3. Design a hypothetical shopping application displays various products offered for sale, and maintains a shopping cart for intended purchases.