**Course: Building Python Web Apps with Django (Backend) and Flutter (Frontend)**

**Introduction**

This course will guide you step-by-step through building full-stack applications using Django as the backend and Flutter as the frontend. You will learn the theoretical background, the practical implementation, and best practices to improve your development workflow.

**Module 1: Understanding Django (Backend)**

**1.1 Introduction to Django**

* What is Django?
* Why use Django?
* Django’s MVT (Model-View-Template) architecture
* Setting up Django in a Virtual Environment

**1.2 Django Folder Structure**

my\_project/

│-- manage.py

│-- my\_project/

│ │-- \_\_init\_\_.py

│ │-- settings.py

│ │-- urls.py

│ │-- wsgi.py

│-- core/

│ │-- \_\_init\_\_.py

│ │-- admin.py

│ │-- apps.py

│ │-- models.py

│ │-- views.py

│ │-- urls.py

│ │-- serializers.py

│-- templates/

│-- static/

│-- db.sqlite3

* Explanation of each file and its purpose

**1.3 Setting Up the Database and Migrations**

* Configuring settings.py for database connection
* Running migrations (python manage.py migrate)
* Creating models in models.py
* Applying migrations (python manage.py makemigrations and python manage.py migrate)

**1.4 Creating Views and URLs**

* What are views in Django?
* Defining views in views.py
* Connecting views to URLs in urls.py
* Using Django’s built-in authentication views

**1.5 Building API Endpoints**

* Creating API views using Django REST Framework (DRF)
* Serializers (serializers.py)
* Using @csrf\_exempt and @require\_http\_methods decorators
* Handling GET and POST requests

**1.6 Handling User Authentication**

* User model and authentication system
* Creating signup and login APIs
* Using Django’s User model and sessions
* Secure authentication with tokens

**1.7 Configuring CORS and API Security**

* Understanding CORS (Cross-Origin Resource Sharing)
* Installing django-cors-headers
* Updating settings.py to allow requests from the frontend
* Debugging common API security issues

**Module 2: Flutter (Frontend)**

**2.1 Introduction to Flutter**

* What is Flutter?
* Why use Flutter?
* Understanding Dart language basics
* Setting up Flutter SDK and VS Code/Android Studio

**2.2 Flutter Folder Structure**

my\_app/

│-- lib/

│ │-- main.dart

│ │-- screens/

│ │ │-- login\_screen.dart

│ │ │-- home\_screen.dart

│ │-- services/

│ │ │-- api\_service.dart

│ │-- widgets/

│ │ │-- custom\_button.dart

│-- pubspec.yaml

* Explanation of each folder and file

**2.3 UI Development in Flutter**

* Widgets and widget tree
* Stateless vs Stateful widgets
* Material design components
* Handling navigation between screens

**2.4 Theming and Styling**

* Changing colors and fonts
* Using ThemeData for global styles
* Creating reusable UI components

**2.5 Handling User Input and Forms**

* Using TextField for input
* Form validation
* Managing state in Flutter

**2.6 Making API Calls from Flutter**

* Using http package to send requests
* Sending JSON data in POST requests
* Handling API responses

**Module 3: Connecting Flutter to Django**

**3.1 Understanding API Communication**

* What is REST API?
* How HTTP requests work (GET, POST, PUT, DELETE)
* JSON serialization and deserialization

**3.2 Setting Up API Calls in Flutter**

* Creating api\_service.dart
* Implementing login and signup requests
* Handling authentication tokens

**3.3 Managing State and API Responses**

* Using setState to update UI after API calls
* Handling success and error responses

**3.4 Navigating Between Screens**

* Redirecting users after login
* Passing parameters between screens (Navigator.pushNamed)
* Redirecting back to login after logout

**3.5 Handling Sessions in Flutter and Django**

* How sessions work in Django
* Storing user credentials locally (SharedPreferences)
* Handling authentication state in Flutter

**Module 4: Deployment and Optimization**

**4.1 Running Django Server**

* Starting the development server (python manage.py runserver)
* Changing ports (python manage.py runserver 0.0.0.0:8000)
* Deploying Django backend (Heroku, AWS, DigitalOcean, etc.)

**4.2 Running Flutter App**

* Running Flutter on Chrome (flutter run -d chrome)
* Building APK for Android (flutter build apk)
* Deploying to Firebase or App Store

**4.3 Debugging and Performance Optimization**

* Debugging Django API errors
* Debugging Flutter UI issues
* Optimizing API calls for performance

**Conclusion**

* Recap of Django backend development
* Recap of Flutter frontend development
* How Django and Flutter communicate
* Next steps for mastering full-stack development

This course provides a beginner-friendly but comprehensive guide to building Python web apps with Django and Flutter. By the end, you will be able to develop, optimize, and deploy full-stack applications. 🚀

**Course: Building Python Web Apps with Django (Backend) and Flutter (Frontend)**

**Introduction**

This course will guide you step-by-step through building full-stack applications using Django as the backend and Flutter as the frontend. You will learn the theoretical background, the practical implementation, and best practices to improve your development workflow.

**Module 1: Understanding Django (Backend)**

**1.1 Introduction to Django**

**What is Django?**

Django is a high-level Python web framework that allows developers to build secure, scalable, and maintainable web applications quickly. It follows the **Model-View-Template (MVT)** architecture, which helps separate concerns in application development.

**Why use Django?**

* **Rapid development**: Built-in tools for authentication, database management, and admin interface.
* **Security**: Protection against SQL injection, cross-site scripting (XSS), and cross-site request forgery (CSRF).
* **Scalability**: Suitable for small to large-scale applications.
* **Batteries included**: Comes with many features like ORM, caching, middleware, and API handling.

**Setting up Django in a Virtual Environment**

1. Install Python (>=3.10 recommended)
2. Create a virtual environment:

python -m venv myenv

1. Activate the environment:
   * Windows:

myenv\Scripts\activate

* + macOS/Linux:

source myenv/bin/activate

1. Install Django:

pip install django

1. Verify installation:

django-admin --version

1. Create a Django project:

django-admin startproject my\_project

1. Run the development server:
2. cd my\_project

python manage.py runserver

This will start the server at http://127.0.0.1:8000/.

**1.2 Django Folder Structure**

my\_project/

│-- manage.py # Command-line utility

│-- my\_project/ # Main project directory

│ │-- \_\_init\_\_.py # Marks this as a Python package

│ │-- settings.py # Project configurations

│ │-- urls.py # URL routing

│ │-- wsgi.py # WSGI entry point for servers

│-- core/ # Django app directory

│ │-- \_\_init\_\_.py # Marks this as a Python package

│ │-- admin.py # Admin interface configurations

│ │-- apps.py # Application settings

│ │-- models.py # Database models

│ │-- views.py # Handles logic and API responses

│ │-- urls.py # Routes URLs to views

│ │-- serializers.py # Converts models to JSON (for APIs)

│-- templates/ # HTML templates (for Django apps)

│-- static/ # CSS, JS, images

│-- db.sqlite3 # Default SQLite database

**Explanation of Key Files:**

* **manage.py**: Used for executing commands (e.g., migrations, running the server).
* **settings.py**: Configuration file for database, installed apps, middleware, etc.
* **urls.py**: Maps URLs to views.
* **models.py**: Defines database models.
* **views.py**: Handles request/response logic.
* **serializers.py**: Converts models into JSON format for APIs (used in Django REST Framework).

**1.3 Setting Up the Database and Migrations**

**Configuring settings.py for Database**

Django uses SQLite by default but supports PostgreSQL, MySQL, and others. To use PostgreSQL, modify settings.py:

DATABASES = {

'default': {

'ENGINE': 'django.db.backends.postgresql',

'NAME': 'mydatabase',

'USER': 'myuser',

'PASSWORD': 'mypassword',

'HOST': 'localhost',

'PORT': '5432',

}

}

**Running Migrations**

1. Apply existing migrations:

python manage.py migrate

1. Define models in models.py:
2. from django.db import models
3. class UserProfile(models.Model):
4. username = models.CharField(max\_length=100, unique=True)
5. email = models.EmailField(unique=True)

date\_joined = models.DateTimeField(auto\_now\_add=True)

1. Create migrations:

python manage.py makemigrations

1. Apply migrations:

python manage.py migrate

**1.4 Creating Views and URLs**

**What are Views in Django?**

Views handle HTTP requests and responses. Example:

from django.http import JsonResponse

def home(request):

return JsonResponse({"message": "Hello, Django!"})

**Connecting Views to URLs**

Modify urls.py:

from django.urls import path

from .views import home

urlpatterns = [

path('', home, name='home'),

]

**1.5 Building API Endpoints with Django REST Framework (DRF)**

**Installing DRF**

pip install djangorestframework

Add to settings.py:

INSTALLED\_APPS = [

'rest\_framework',

]

**Creating a Serializer**

Modify serializers.py:

from rest\_framework import serializers

from .models import UserProfile

class UserProfileSerializer(serializers.ModelSerializer):

class Meta:

model = UserProfile

fields = '\_\_all\_\_'

**Creating an API View**

Modify views.py:

from rest\_framework.decorators import api\_view

from rest\_framework.response import Response

from .models import UserProfile

from .serializers import UserProfileSerializer

@api\_view(['GET'])

def get\_users(request):

users = UserProfile.objects.all()

serializer = UserProfileSerializer(users, many=True)

return Response(serializer.data)

**Adding API URL**

Modify urls.py:

from django.urls import path

from .views import get\_users

urlpatterns = [

path('api/users/', get\_users, name='get\_users'),

]

**1.6 Handling User Authentication**

* Using Django’s built-in authentication system (User model).
* Creating login and signup APIs.
* Using Django sessions or tokens for authentication.

**1.7 Configuring CORS and API Security**

**Installing CORS Headers**

pip install django-cors-headers

Add to settings.py:

INSTALLED\_APPS += ['corsheaders']

MIDDLEWARE += ['corsheaders.middleware.CorsMiddleware']

CORS\_ALLOW\_ALL\_ORIGINS = True # Allow requests from frontend

Next, we will cover **Module 2: Flutter (Frontend)** where we will design the UI and connect to Django APIs. 🚀

**Module 2: Flutter (Frontend)**

**2.1 Introduction to Flutter**

**What is Flutter?**

Flutter is an open-source UI toolkit created by Google for building natively compiled applications for mobile, web, and desktop from a single codebase. It uses the **Dart** programming language and provides a rich set of pre-designed widgets for building modern interfaces.

**Why use Flutter?**

* **Cross-platform development**: Write one codebase for multiple platforms (iOS, Android, Web, Desktop).
* **Fast development**: Hot-reload feature allows instant preview of changes.
* **Expressive UI**: Customizable widgets for beautiful interfaces.
* **High performance**: Uses Dart’s compiled nature and Skia rendering engine.

**Understanding Dart Language Basics**

* Dart is an object-oriented language with syntax similar to Java and JavaScript.
* Key concepts include:
  + Variables and data types (int, double, String, bool, List, Map)
  + Functions and methods
  + Classes and objects
  + Asynchronous programming (async/await for handling API calls)

**Setting Up Flutter SDK and Development Environment**

1. Download and install the Flutter SDK from [flutter.dev](https://flutter.dev/).
2. Install a code editor (VS Code or Android Studio recommended).
3. Set up Flutter in the terminal:

flutter doctor

This checks for missing dependencies.

1. Create a new Flutter project:

flutter create my\_app

1. Run the app:
2. cd my\_app

flutter run

**2.2 Flutter Folder Structure**

my\_app/

│-- lib/

│ │-- main.dart

│ │-- screens/

│ │ │-- login\_screen.dart

│ │ │-- home\_screen.dart

│ │-- services/

│ │ │-- api\_service.dart

│ │-- widgets/

│ │ │-- custom\_button.dart

│-- pubspec.yaml

**Explanation of Each Folder and File:**

* **main.dart**: Entry point of the application.
* **screens/**: Contains UI screens (e.g., Login, Home).
* **services/**: Handles API calls and backend communication.
* **widgets/**: Contains reusable UI components.
* **pubspec.yaml**: Manages dependencies and assets.

**2.3 UI Development in Flutter**

**2.3 UI Development in Flutter**

**Flutter’s UI is built using widgets, which are the building blocks of the application. The UI follows a declarative approach, meaning the UI is rebuilt when the state changes.**

**1. Widgets and Widget Tree**

* **Widgets are the fundamental UI components in Flutter. Everything in Flutter is a widget, from buttons to entire screens.**
* **Flutter organizes widgets into a widget tree, where parent widgets contain child widgets, forming a hierarchy.**
* **The widget tree updates dynamically when the app's state changes.**

**2. Stateless vs Stateful Widgets**

* **StatelessWidget: A widget that does not change once it is built. It is immutable.**
* **StatefulWidget: A widget that can change over time due to user interaction or state updates.**

**Example:**

**dart**

**CopyEdit**

**class MyStatelessWidget extends StatelessWidget {**

**@override**

**Widget build(BuildContext context) {**

**return Text("I never change!");**

**}**

**}**

**class MyStatefulWidget extends StatefulWidget {**

**@override**

**\_MyStatefulWidgetState createState() => \_MyStatefulWidgetState();**

**}**

**class \_MyStatefulWidgetState extends State<MyStatefulWidget> {**

**int counter = 0;**

**void \_increment() {**

**setState(() {**

**counter++;**

**});**

**}**

**@override**

**Widget build(BuildContext context) {**

**return Column(**

**children: [**

**Text("Counter: $counter"),**

**ElevatedButton(**

**onPressed: \_increment,**

**child: Text("Increment"),**

**),**

**],**

**);**

**}**

**}**

**In this example:**

* **MyStatelessWidget never changes.**
* **MyStatefulWidget updates the counter when the button is pressed.**

**3. Material Design Components**

**Flutter provides Material Design components to ensure a modern and consistent UI. Some commonly used components include:**

* **Scaffold (Basic structure with AppBar, FloatingActionButton, Drawer)**
* **AppBar (Toolbar at the top of the screen)**
* **ElevatedButton (Button with elevation effect)**
* **TextField (Input field for user data)**
* **ListView (Scrollable list of items)**

**Example:**

**dart**

**CopyEdit**

**Scaffold(**

**appBar: AppBar(title: Text("Material Design Example")),**

**body: Center(child: ElevatedButton(onPressed: () {}, child: Text("Click Me"))),**

**);**

**4. Handling Navigation Between Screens**

**Flutter provides a Navigator to manage screen transitions.**

* **Navigator.push() to navigate to a new screen.**
* **Navigator.pop() to go back to the previous screen.**

**Example:**

**dart**

**CopyEdit**

**// First Screen**

**class FirstScreen extends StatelessWidget {**

**@override**

**Widget build(BuildContext context) {**

**return Scaffold(**

**appBar: AppBar(title: Text("First Screen")),**

**body: Center(**

**child: ElevatedButton(**

**onPressed: () {**

**Navigator.push(**

**context,**

**MaterialPageRoute(builder: (context) => SecondScreen()),**

**);**

**},**

**child: Text("Go to Second Screen"),**

**),**

**),**

**);**

**}**

**}**

**// Second Screen**

**class SecondScreen extends StatelessWidget {**

**@override**

**Widget build(BuildContext context) {**

**return Scaffold(**

**appBar: AppBar(title: Text("Second Screen")),**

**body: Center(**

**child: ElevatedButton(**

**onPressed: () {**

**Navigator.pop(context);**

**},**

**child: Text("Go Back"),**

**),**

**),**

**);**

**}**

**}**

**This creates two screens, where a button on the first screen navigates to the second, and the second screen has a button to return.**

**2.4 Theming and Styling**

**1. Changing Colors and Fonts**

**Flutter allows you to customize colors and fonts directly within widgets using the style property.**

**Example: Styling Text and Containers**

**dart**

**CopyEdit**

**Text(**

**"Hello, Flutter!",**

**style: TextStyle(**

**fontSize: 24,**

**fontWeight: FontWeight.bold,**

**color: Colors.blue,**

**fontFamily: 'Roboto', // Ensure the font is added in pubspec.yaml**

**),**

**),**

**Container(**

**padding: EdgeInsets.all(16),**

**decoration: BoxDecoration(**

**color: Colors.blue.shade100,**

**borderRadius: BorderRadius.circular(10),**

**),**

**child: Text("Styled Container"),**

**),**

**Here, we:  
✅ Set custom text color, font size, and font family.  
✅ Apply a background color and rounded borders to a container.**

**2. Using ThemeData for Global Styles**

**Instead of applying styles to individual widgets, we can define a global theme using ThemeData, which helps maintain consistency across the app.**

**Example: Defining a Global Theme**

**Modify the MaterialApp to include a global theme:**

**dart**

**CopyEdit**

**MaterialApp(**

**theme: ThemeData(**

**primarySwatch: Colors.teal, // Primary app color**

**textTheme: TextTheme(**

**bodyMedium: TextStyle(fontSize: 18, color: Colors.black87),**

**titleLarge: TextStyle(fontSize: 24, fontWeight: FontWeight.bold),**

**),**

**elevatedButtonTheme: ElevatedButtonThemeData(**

**style: ButtonStyle(**

**backgroundColor: MaterialStateProperty.all(Colors.teal),**

**),**

**),**

**),**

**home: HomeScreen(),**

**);**

**Now, all ElevatedButton widgets in the app will use the defined style, and text widgets will follow the global textTheme.**

**Example: Using the Theme in Widgets**

**dart**

**CopyEdit**

**Text(**

**"This is themed text",**

**style: Theme.of(context).textTheme.titleLarge,**

**),**

**ElevatedButton(**

**onPressed: () {},**

**child: Text("Click Me"),**

**),**

**Here, the text and button automatically follow the global theme settings.**

**3. Creating Reusable UI Components**

**To keep the codebase clean and reusable, we can create custom UI components.**

**Example: Custom Button Widget**

**Instead of defining button styles in every screen, we create a reusable component:**

**dart**

**CopyEdit**

**class CustomButton extends StatelessWidget {**

**final String text;**

**final VoidCallback onPressed;**

**const CustomButton({required this.text, required this.onPressed, Key? key}) : super(key: key);**

**@override**

**Widget build(BuildContext context) {**

**return ElevatedButton(**

**onPressed: onPressed,**

**style: ElevatedButton.styleFrom(**

**padding: EdgeInsets.symmetric(horizontal: 20, vertical: 12),**

**textStyle: TextStyle(fontSize: 18),**

**),**

**child: Text(text),**

**);**

**}**

**}**

**Using the Custom Button**

**dart**

**CopyEdit**

**CustomButton(**

**text: "Reusable Button",**

**onPressed: () {**

**print("Button Clicked!");**

**},**

**),**

**Now, we can reuse CustomButton across the app instead of repeating styling code.**

**Summary**

**✅ Inline styling allows quick customization for individual widgets.  
✅ ThemeData enables global styling for consistency.  
✅ Reusable components improve maintainability and reduce code duplication.**

**2.5 Handling User Input and Forms**

**User input is a core part of any app. In Flutter, we use widgets like TextField to accept input, implement form validation, and manage state effectively.**

**2.5.1 Using TextField for Input**

**The TextField widget allows users to enter text. It’s commonly used for search bars, login forms, and user feedback.**

**Basic Example: A Simple Text Input**

**dart**

**CopyEdit**

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

**void main() {**

**runApp(const MyApp());**

**}**

**class MyApp extends StatelessWidget {**

**const MyApp({Key? key}) : super(key: key);**

**@override**

**Widget build(BuildContext context) {**

**return MaterialApp(**

**home: TextFieldExample(),**

**);**

**}**

**}**

**class TextFieldExample extends StatelessWidget {**

**TextFieldExample({Key? key}) : super(key: key);**

**final TextEditingController \_controller = TextEditingController();**

**@override**

**Widget build(BuildContext context) {**

**return Scaffold(**

**appBar: AppBar(title: const Text("User Input Example")),**

**body: Padding(**

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

**child: Column(**

**children: [**

**TextField(**

**controller: \_controller, // Captures input**

**decoration: const InputDecoration(**

**labelText: "Enter your name",**

**border: OutlineInputBorder(),**

**),**

**),**

**const SizedBox(height: 20),**

**ElevatedButton(**

**onPressed: () {**

**print("User entered: ${\_controller.text}");**

**},**

**child: const Text("Submit"),**

**),**

**],**

**),**

**),**

**);**

**}**

**}**

**Key Concepts in This Example**

**✅ TextEditingController is used to read input from TextField.  
✅ decoration adds a label and border for better UI.  
✅ The button prints the entered text when clicked.**

**2.5.2 Form Validation**

**For validating input (e.g., checking if an email is valid), we use Form and TextFormField.**

**Example: Validating User Input**

**dart**

**CopyEdit**

**class FormValidationExample extends StatefulWidget {**

**const FormValidationExample({Key? key}) : super(key: key);**

**@override**

**\_FormValidationExampleState createState() => \_FormValidationExampleState();**

**}**

**class \_FormValidationExampleState extends State<FormValidationExample> {**

**final \_formKey = GlobalKey<FormState>();**

**final TextEditingController \_emailController = TextEditingController();**

**@override**

**Widget build(BuildContext context) {**

**return Scaffold(**

**appBar: AppBar(title: const Text("Form Validation")),**

**body: Padding(**

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

**child: Form(**

**key: \_formKey, // Tracks form state**

**child: Column(**

**children: [**

**TextFormField(**

**controller: \_emailController,**

**decoration: const InputDecoration(**

**labelText: "Email",**

**border: OutlineInputBorder(),**

**),**

**validator: (value) {**

**if (value == null || value.isEmpty) {**

**return "Please enter an email";**

**} else if (!value.contains("@")) {**

**return "Enter a valid email";**

**}**

**return null;**

**},**

**),**

**const SizedBox(height: 20),**

**ElevatedButton(**

**onPressed: () {**

**if (\_formKey.currentState!.validate()) {**

**print("Valid email: ${\_emailController.text}");**

**}**

**},**

**child: const Text("Submit"),**

**),**

**],**

**),**

**),**

**),**

**);**

**}**

**}**

**How This Works**

**✅ Form groups fields together and tracks validation.  
✅ GlobalKey<FormState> helps manage the form state.  
✅ TextFormField includes a validator function to check input validity.**

**2.5.3 Managing State in Flutter**

**Handling user input requires managing state. In Flutter, state is used to store and update values dynamically.**

**Types of State Management**

1. **Local State: Managed within a single widget using StatefulWidget.**
2. **App-Wide State: Managed using providers like Provider, Riverpod, or Bloc.**

**Example: Managing Input State Locally**

**dart**

**CopyEdit**

**class StatefulTextFieldExample extends StatefulWidget {**

**const StatefulTextFieldExample({Key? key}) : super(key: key);**

**@override**

**\_StatefulTextFieldExampleState createState() => \_StatefulTextFieldExampleState();**

**}**

**class \_StatefulTextFieldExampleState extends State<StatefulTextFieldExample> {**

**String \_name = "";**

**@override**

**Widget build(BuildContext context) {**

**return Scaffold(**

**appBar: AppBar(title: const Text("Stateful Input Example")),**

**body: Padding(**

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

**child: Column(**

**children: [**

**TextField(**

**onChanged: (value) {**

**setState(() {**

**\_name = value; // Updates state when text changes**

**});**

**},**

**decoration: const InputDecoration(**

**labelText: "Enter your name",**

**border: OutlineInputBorder(),**

**),**

**),**

**const SizedBox(height: 20),**

**Text("Hello, $\_name"), // Displays updated state**

**],**

**),**

**),**

**);**

**}**

**}**

**How This Works**

**✅ setState() updates \_name whenever the user types.  
✅ The Text widget updates dynamically as the user types.**

**Summary**

| **Concept** | **Key Points** |
| --- | --- |
| **TextField Input** | **Uses TextEditingController to capture user input.** |
| **Form Validation** | **Form and TextFormField validate input before submission.** |
| **Managing State** | **setState() updates UI dynamically.** |

**This section covers essential concepts for handling user input in Flutter. The next section can explore Advanced State Management techniques using Provider or Bloc.**

**2.6 Making API Calls from Flutter**

* **Using http package to send requests**
* **Sending JSON data in POST requests**
* **Handling API responses**

In modern mobile applications, communicating with remote servers is essential. Flutter allows you to easily make HTTP requests using the http package. This section will cover how to send requests, send JSON data, and handle responses.

**2.6.1 Using http Package to Send Requests**

To make HTTP requests in Flutter, the http package is commonly used. You can install it by adding the following to your pubspec.yaml file:

yaml

CopyEdit

dependencies:

http: ^0.14.0

**Basic Example: Sending a GET Request**

The following example demonstrates how to send a simple GET request to fetch data from an API.

dart

CopyEdit

import 'package:flutter/material.dart';

import 'package:http/http.dart' as http;

import 'dart:convert';

void main() {

runApp(const MyApp());

}

class MyApp extends StatelessWidget {

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

@override

Widget build(BuildContext context) {

return MaterialApp(

home: FetchDataScreen(),

);

}

}

class FetchDataScreen extends StatefulWidget {

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

@override

\_FetchDataScreenState createState() => \_FetchDataScreenState();

}

class \_FetchDataScreenState extends State<FetchDataScreen> {

late String data = "Loading...";

@override

void initState() {

super.initState();

fetchData();

}

Future<void> fetchData() async {

final response = await http.get(Uri.parse("https://jsonplaceholder.typicode.com/posts/1"));

if (response.statusCode == 200) {

setState(() {

data = jsonDecode(response.body)['title']; // Parsing the response

});

} else {

setState(() {

data = "Failed to load data";

});

}

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text("API Data Fetching")),

body: Center(child: Text(data)),

);

}

}

**Explanation of the Code**

1. **GET Request**: The http.get() method sends a GET request to the URL.
2. **Response Handling**: If the status code is 200 (success), we decode the JSON response and extract the title.
3. **Displaying Data**: The response data is displayed using a Text widget in the build() method.

**2.6.2 Sending JSON Data in POST Requests**

Often, you need to send data to a server via POST requests, especially when submitting forms. In this case, you can send data as a JSON object.

**Example: Sending Data with POST Request**

Here’s how you can send a POST request with JSON data:

dart

CopyEdit

import 'package:flutter/material.dart';

import 'package:http/http.dart' as http;

import 'dart:convert';

void main() {

runApp(const MyApp());

}

class MyApp extends StatelessWidget {

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

@override

Widget build(BuildContext context) {

return MaterialApp(

home: PostDataScreen(),

);

}

}

class PostDataScreen extends StatefulWidget {

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

@override

\_PostDataScreenState createState() => \_PostDataScreenState();

}

class \_PostDataScreenState extends State<PostDataScreen> {

String responseMessage = "Submitting...";

Future<void> sendPostRequest() async {

final url = Uri.parse("https://jsonplaceholder.typicode.com/posts");

final response = await http.post(

url,

headers: {"Content-Type": "application/json"},

body: jsonEncode({

"title": "foo",

"body": "bar",

"userId": 1,

}),

);

if (response.statusCode == 201) {

setState(() {

responseMessage = "Data Submitted Successfully!";

});

} else {

setState(() {

responseMessage = "Failed to submit data.";

});

}

}

@override

Widget build(BuildContext context) {

return Scaffold(

appBar: AppBar(title: const Text("POST Request Example")),

body: Center(

child: Column(

mainAxisAlignment: MainAxisAlignment.center,

children: [

ElevatedButton(

onPressed: sendPostRequest,

child: const Text("Submit Data"),

),

const SizedBox(height: 20),

Text(responseMessage),

],

),

),

);

}

}

**Explanation of the Code**

1. **POST Request**: We use http.post() to send a POST request to the server.
2. **Headers**: The Content-Type header specifies that the data being sent is in JSON format.
3. **Body**: The body is encoded using jsonEncode() to send the data as a JSON object.
4. **Response**: If the server responds with status code 201 (Created), we update the UI to indicate successful submission.

**2.6.3 Handling API Responses**

After sending a request, it’s important to handle the response, especially errors.

**Handling Errors**

In real-world scenarios, network calls might fail, and you need to handle these errors gracefully.

dart

CopyEdit

Future<void> fetchData() async {

try {

final response = await http.get(Uri.parse("https://jsonplaceholder.typicode.com/posts/1"));

if (response.statusCode == 200) {

setState(() {

data = jsonDecode(response.body)['title'];

});

} else {

setState(() {

data = "Failed to load data. Status Code: ${response.statusCode}";

});

}

} catch (e) {

setState(() {

data = "Error: $e"; // Handling unexpected errors

});

}

}

**Key Concepts for Handling API Responses:**

1. **Error Handling**: Use try-catch blocks to handle any network issues.
2. **Status Codes**: Always check the response status code (200 for success, 404 for not found, etc.).
3. **Parsing Responses**: Use jsonDecode() to parse JSON responses into usable data.
4. **User Feedback**: Always give feedback on the success or failure of API requests.

**Summary: Key Points**

| **Topic** | **Key Concept** |
| --- | --- |
| **GET Requests** | Use http.get() to fetch data from an API. |
| **POST Requests** | Send data using http.post(), with JSON. |
| **Handling Errors** | Use try-catch for network errors. |
| **Parsing Responses** | Use jsonDecode() to parse JSON responses. |

**Next Steps**

* Once you have a working API call, you can integrate this with your backend (like Django) for real data fetching and submission.
* You can also explore **authentication** for secure API calls (e.g., with tokens).

With this, your Flutter app can send data to and receive data from a Django backend (or any other backend) to build dynamic, data-driven applications.

**Module 3: Connecting Flutter to Django**

This module covers how to integrate a Flutter app with a Django backend using RESTful APIs, manage authentication, handle state, and navigate between screens.

**3.1 Understanding API Communication**

**What is REST API?**

A **REST API (Representational State Transfer API)** is a way for a client (Flutter app) to communicate with a server (Django backend) using standard HTTP methods. REST APIs send and receive data in JSON format, making it easy to integrate with mobile and web apps.

**How HTTP Requests Work (GET, POST, PUT, DELETE)**

| **Method** | **Description** | **Example Use Case** |
| --- | --- | --- |
| **GET** | Retrieves data from the server | Fetching user profiles |
| **POST** | Sends data to the server | User registration |
| **PUT** | Updates existing data | Updating a user’s profile |
| **DELETE** | Removes data from the server | Deleting an account |

**Example: Making a GET Request in Flutter**

dart

CopyEdit

import 'package:http/http.dart' as http;

import 'dart:convert';

Future<void> fetchUser() async {

final response = await http.get(Uri.parse("https://example.com/api/user/1"));

if (response.statusCode == 200) {

print(jsonDecode(response.body)); // Prints the user data

} else {

print("Failed to fetch data");

}

}

**JSON Serialization and Deserialization**

* **Serialization**: Converting a Dart object into JSON to send to the server.
* **Deserialization**: Converting a JSON response into a Dart object.

**Example: Parsing JSON Response in Flutter**

dart

CopyEdit

class User {

final String name;

final String email;

User({required this.name, required this.email});

factory User.fromJson(Map<String, dynamic> json) {

return User(

name: json['name'],

email: json['email'],

);

}

}

**3.2 Setting Up API Calls in Flutter**

**Creating api\_service.dart**

A separate Dart file for handling API calls makes the code modular and maintainable.

**Example: api\_service.dart**

dart

CopyEdit

import 'package:http/http.dart' as http;

import 'dart:convert';

class ApiService {

static const String baseUrl = "https://example.com/api/";

Future<Map<String, dynamic>> login(String email, String password) async {

final response = await http.post(

Uri.parse("${baseUrl}login/"),

headers: {"Content-Type": "application/json"},

body: jsonEncode({"email": email, "password": password}),

);

return jsonDecode(response.body);

}

}

**Implementing Login and Signup Requests**

dart

CopyEdit

Future<void> loginUser() async {

final api = ApiService();

final response = await api.login("user@example.com", "password123");

if (response['success']) {

print("Login successful!");

} else {

print("Login failed: ${response['message']}");

}

}

**Handling Authentication Tokens**

* Django typically returns a token (JWT or session-based) upon login.
* Flutter should store and send this token with each request.

**3.3 Managing State and API Responses**

**Using setState to Update UI After API Calls**

When API responses change the UI (e.g., fetching user info), use setState.

**Example**

dart

CopyEdit

class HomePage extends StatefulWidget {

@override

\_HomePageState createState() => \_HomePageState();

}

class \_HomePageState extends State<HomePage> {

String data = "Loading...";

void fetchData() async {

final api = ApiService();

final response = await api.login("user@example.com", "password123");

setState(() {

data = response['message'];

});

}

@override

Widget build(BuildContext context) {

return Scaffold(

body: Center(child: Text(data)),

);

}

}

**Handling Success and Error Responses**

* Always check for response.statusCode before processing data.
* Use try-catch blocks to handle network failures.

dart

CopyEdit

try {

final response = await http.get(Uri.parse("https://example.com/api/data"));

if (response.statusCode == 200) {

print("Data loaded successfully!");

} else {

print("Error: ${response.reasonPhrase}");

}

} catch (e) {

print("Network error: $e");

}

**3.4 Navigating Between Screens**

**Redirecting Users After Login**

dart

CopyEdit

if (response['success']) {

Navigator.pushReplacementNamed(context, "/home");

}

**Passing Parameters Between Screens (Navigator.pushNamed)**

dart

CopyEdit

Navigator.pushNamed(context, "/profile", arguments: {"userId": 1});

On the receiving page:

dart

CopyEdit

final args = ModalRoute.of(context)!.settings.arguments as Map;

int userId = args["userId"];

**Redirecting Back to Login After Logout**

dart

CopyEdit

void logout(BuildContext context) {

Navigator.pushNamedAndRemoveUntil(context, "/login", (route) => false);

}

**3.5 Handling Sessions in Flutter and Django**

**How Sessions Work in Django**

Django uses session cookies to track users. A session ID is stored in the database and linked to the user.

Example Django session setup:

python

CopyEdit

from django.contrib.sessions.models import Session

# Get session data

session = Session.objects.get(session\_key="some\_key")

**Storing User Credentials Locally (SharedPreferences)**

Flutter can store tokens or session data using SharedPreferences.

**Example: Saving and Retrieving Tokens**

dart

CopyEdit

import 'package:shared\_preferences/shared\_preferences.dart';

Future<void> saveToken(String token) async {

final prefs = await SharedPreferences.getInstance();

await prefs.setString("auth\_token", token);

}

Future<String?> getToken() async {

final prefs = await SharedPreferences.getInstance();

return prefs.getString("auth\_token");

}

**Handling Authentication State in Flutter**

* Check if a user is logged in before navigating.
* Redirect users accordingly.

dart

CopyEdit

void checkLoginStatus() async {

String? token = await getToken();

if (token != null) {

Navigator.pushReplacementNamed(context, "/home");

} else {

Navigator.pushReplacementNamed(context, "/login");

}

}

**Conclusion**

By the end of this module, students should be able to: ✅ Understand how Flutter communicates with Django through REST APIs.  
✅ Implement API requests for login, signup, and fetching data.  
✅ Manage app state after API calls.  
✅ Navigate between screens and pass data.  
✅ Store authentication tokens and manage user sessions.