Lab: Calling an API Back-End in Flutter

Estimated time: 30 minutes

Welcome to this hands-on lab where you will learn how to call an API back-end in Flutter, handle responses, and update your app's UI with the retrieved data. This lab will help you understand asynchronous programming in Flutter and how to use the http package to make HTTP requests.

Objectives

After completing this lab, you will be able to:

- Understand how to use the http package to make HTTP requests
- Implement asynchronous programming using Future and async/await in Flutter
- Parse JSON data and map it to Dart models
- Update the Flutter UI with dynamic data fetched from an API

About Cloud IDE

Running the lab

This lab is designed to be completed using a Cloud IDE environment. You will be writing and executing Dart code directly in the IDE.

About Skills Network Cloud IDE

Skills Network Cloud IDE (based on Theia and Docker) provides an environment for hands-on labs for course and project-related labs. It is an open-source integrated development environment (IDE).

Important Notice about this lab environment

Please be aware that sessions for this lab environment are not persisted. Every time you connect to this lab, a new environment is created for you. Any data you may have saved in the earlier session would get lost. It's a good idea to plan and complete these labs in a single session, to avoid losing your data.

Key terms

- Application programming interface (API): Allows different software applications to communicate with each other.
- http package: A Flutter package for making HTTP requests.
- Asynchronous programming: A way of programming that allows multiple tasks to run concurrently, such as waiting for data to be fetched from a server.
- JSON (JavaScript Object Notation): A lightweight data format used for transmitting data between a server and a client.

Prerequisites

Before starting this lab, ensure you have the following setup in your Cloud IDE:

1. Create a new Flutter project:

```
flutter create flutter_api_lab
cd flutter_api_lab
```

2. Open the `lib/main.dart` file in your project.

Step 1: Set up the http package and create a model

1. Open your `pubspec.yaml` file and add the http package under the dependencies section:

```
dependencies:
  flutter:
    sdk: flutter
  http: ^0.13.3
```

- The http package allows you to send HTTP requests to interact with web services.

2. Save the `pubspec.yaml` file and install the plugin by running:

```
flutter pub get
```

3. Create a new file `lib/post_model.dart`. Add the following code for a model class to represent the JSON data structure:

```
import 'dart:convert';  // Import for JSON decoding
class Post {
  final int id;
  final String title;
  final String body;
  Post({required this.id, required this.title, required this.body});
  factory Post.fromJson(Map<String, dynamic> json) {
    return Post(
      id: json['id'],
      title: json['title'],
      body: json['body'],
    );
  }
}
```

- The `Post` class represents a post object with `id`, `title`, and `body` fields.
- The `fromJson` factory method parses JSON data into a `Post` object.

Step 2: Make the API call and handle the response

1. Open your `lib/main.dart` file and import the necessary packages:

```
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
import 'dart:convert';
import 'post_model.dart';
```

- Import the `http` package for HTTP requests, `dart:convert` for JSON decoding, and `post_model.dart` for the model class.

2. Create a function to fetch data from the API:

```
Future<List<Post>> fetchPosts() async {
  final response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts'));
  if (response.statusCode == 200) {
    List jsonResponse = json.decode(response.body);
    return jsonResponse.map((post) => Post.fromJson(post)).toList();
  } else {
    throw Exception('Failed to load posts');
  }
}
```

- The `fetchPosts` function sends an HTTP GET request to the API endpoint.
- If the request is successful (`statusCode` 200), it decodes the JSON response and maps it to a list of `Post` objects.

3. Create a StatefulWidget to display the data inside the main function:

```
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter API Lab',
      home: const PostList(),
    );
  }
}
class PostList extends StatefulWidget {
  const PostList({super.key});
  @override
  _PostListState createState() => _PostListState();
}
class _PostListState extends State<PostList> {
  late Future<List<Post>> futurePosts;
  @override
  void initState() {
    super.initState();
    futurePosts = fetchPosts();
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('API Data'),
      ),
      body: FutureBuilder<List<Post>>(
        future: futurePosts,
        builder: (context, snapshot) {
          if (snapshot.connectionState == ConnectionState.waiting) {
            return const Center(child: CircularProgressIndicator());
          } else if (snapshot.hasError) {
            return Center(child: Text('Error: ${snapshot.error}'));
          } else if (snapshot.hasData) {
            return ListView.builder(
              itemCount: snapshot.data!.length,
              itemBuilder: (context, index) {
                return ListTile(
                  title: Text(snapshot.data![index].title),
                  subtitle: Text(snapshot.data![index].body),
                );
              },
            );
          } else {
            return const Center(child: Text('No data available'));
          }
        },
      ),
    );
  }
}
```

- `void main()`: The main entry point of the Flutter app. Calls `runApp(MyApp())` to start the application with the `MyApp` widget.
- `runApp(const MyApp())`: Creates and attaches `MyApp` to the screen, which sets up the basic app structure.
- `class MyApp extends StatelessWidget`: The root widget of the app. It doesn't change state; it simply defines the app's structure.
  - `MaterialApp`: Provides the core app features and sets the `home` to `PostList`, which is the main screen where data will be displayed.
- `class PostList extends StatefulWidget`: A stateful widget that allows dynamic content. Here, it is used to display data fetched from an API.
  - `@override _PostListState createState()`: Creates the mutable state for `PostList` to manage data fetching and display.
- `class _PostListState extends State<PostList>`: Manages the state for `PostList`.
  - `futurePosts = fetchPosts()`: Initiates the API call to fetch posts when the widget is initialized.
  - `FutureBuilder<List<Post>>`: Reacts to the future (`futurePosts`) and rebuilds the UI depending on the state of the data fetching (loading, error, or success).
    - Shows a loading indicator (`CircularProgressIndicator`) while waiting.
    - Displays an error message if the API call fails.
    - Uses `ListView.builder` to dynamically generate a scrollable list of posts when data is successfully fetched.

Step 3: Run the app

Flutter provides `hot reload` that makes it very easy to test the code as you are developing it. That funcationality however does not work in the Cloud IDE. We have developed a `hot_reload.sh` script that essentially does the same thing. Please follow these setps to run the flutter application:

1. Change to the home directory

   ```
   cd /home/project
   ```

2. Set the `PROJECT_DIR` variable. This should be set to the `lib` directory of your flutter app folder.

   ```
   export PROJECT_DIR=/home/project/flutter_api_lab/lib
   ```

3. Get the script and save to the `/home/project` folder.

   ```
   wget -O /home/project/hot_reload.sh https://cf-courses-data.s3.us.cloud-object-storage.appdomain.cloud/S3cw83zxbxBhfixSSMK1Uw/hot-reload.sh
   ```

4. Make the script executable by using the `chmod` command.

   ```
   chmod +x ./hot_reload.sh
   ```

5. Run the command.

   ```
   ./hot_reload.sh
   ```

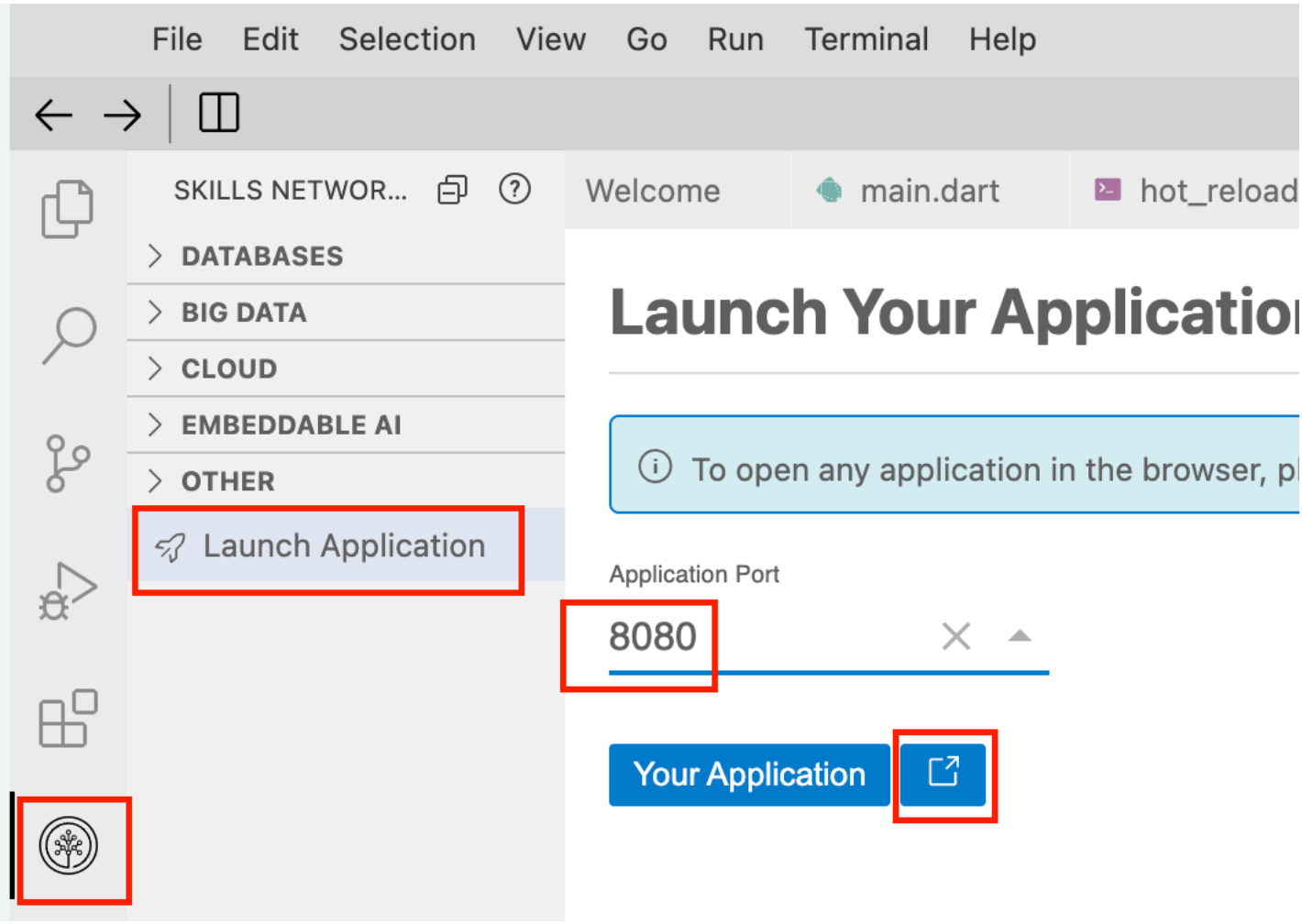
   You should see output like this:

   ```
   theia@theia-captainfedo1:/home/project$ ./hot_reload.sh
   Using PROJECT_DIR: /home/project/flutter_api_lab/lib
   inotify-tools is already installed.
   lsof is already installed.
   Starting Flutter web server...
   Port 8080 is free.
   ```

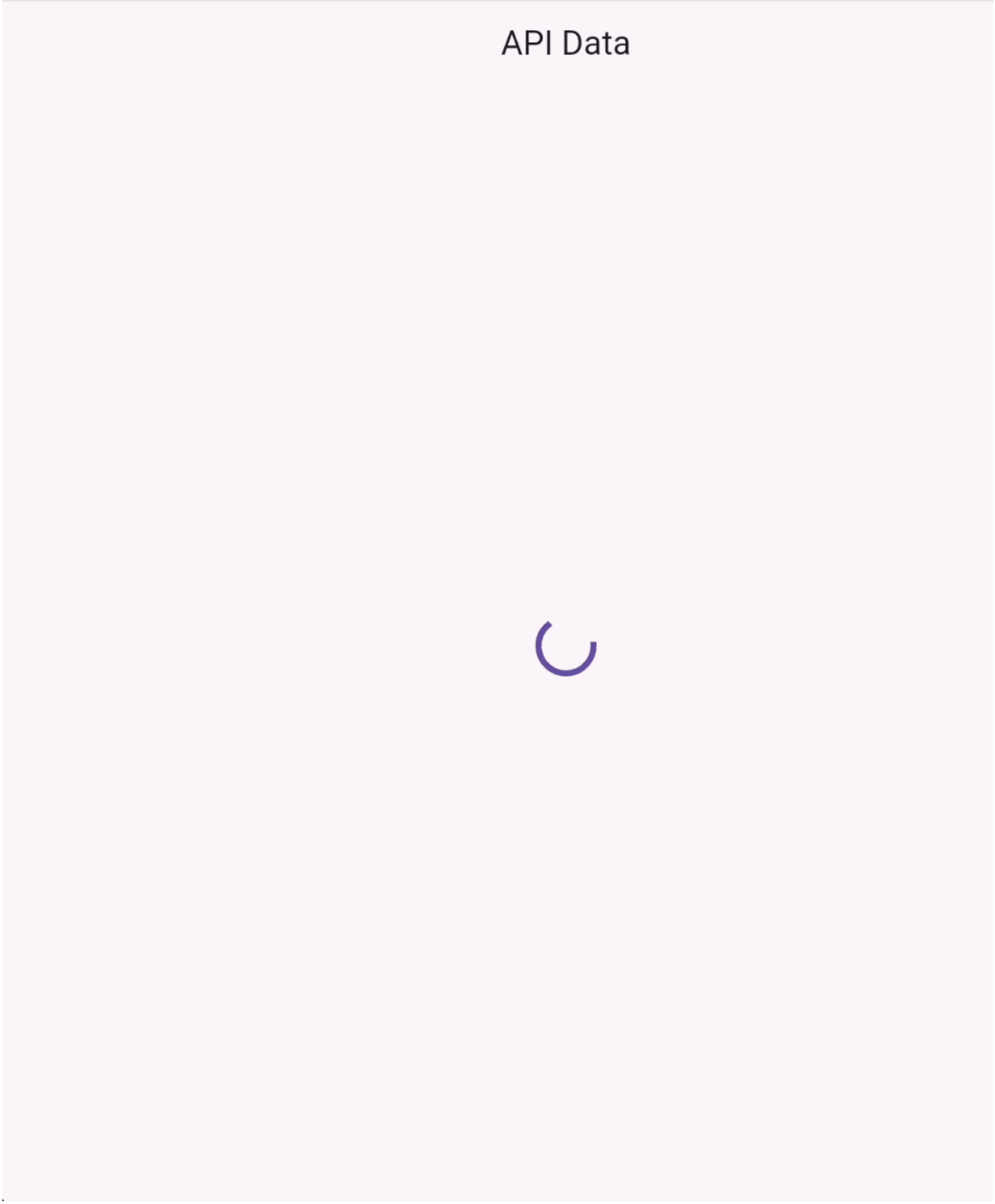
```
Compiling lib/main.dart for the Web...                             984ms
✓ Built build/web
Flutter server started with PID 17675
Setting up watches.  Beware: since -r was given, this may take a while!
Watches established.
Serving HTTP on 0.0.0.0 port 8080 (http://0.0.0.0:8080/) ...
```

6. Launch the application

- open the `Skills Network Toolbox` in the let bar of the Cloud IDE.
- click on `Launch Application` button
- enter `8080` as the port
- select the `launch in new window` to open the applicaiton in a new tab



You will see a loading icon initially as the data is loaded from the remote URL (acting as back end here):



Once data has been loaded, you will see it appear on the screen:

API Data

sunt aut facere repellat provident occaecati excepturi optio reprehen

quia et suscipit
suscipit recusandae consequuntur expedita et cum
reprehenderit molestiae ut ut quas totam
nostrum rerum est autem sunt rem eveniet architecto

qui est esse

est rerum tempore vitae
sequi sint nihil reprehenderit dolor beatae ea dolores neque
fugiat blanditiis voluptate porro vel nihil molestiae ut reiciendis
qui aperiam non debitis possimus qui neque nisi nulla

ea molestias quasi exercitationem repellat qui ipsa sit aut

et iusto sed quo iure
voluptatem occaecati omnis eligendi aut ad
voluptatem doloribus vel accusantium quis pariatur
molestiae porro eius odio et labore et velit aut

eum et est occaecati

ullam et saepe reiciendis voluptatem adipisci
sit amet autem assumenda provident rerum culpa
quis hic commodi nesciunt rem tenetur doloremque ipsam iure
quis sunt voluptatem rerum illo velit

nesciunt quas odio

repudiandae veniam quaerat sunt sed
alias aut fugiat sit autem sed est
voluptatem omnis possimus esse voluptatibus quis
est aut tenetur dolor neque

dolorem eum magni eos aperiam quia

ut aspernatur corporis harum nihil quis provident sequi
mollitia nobis aliquid molestiae

perspiciatis et ea nemo ab reprehenderit accusantium quas
voluptate dolores velit et doloremque molestiae

Step 4: Complete Code

5. Here is the complete main file:

```
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
import 'dart:convert';
import 'post_model.dart';
Future<List<Post>> fetchPosts() async {
  final response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts'));
  if (response.statusCode == 200) {
    List jsonResponse = json.decode(response.body);
    return jsonResponse.map((post) => Post.fromJson(post)).toList();
  } else {
    throw Exception('Failed to load posts');
  }
}
void main() {
  runApp(const MyApp());
}
class MyApp extends StatelessWidget {
  const MyApp({super.key});
  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Flutter API Lab',
      home: const PostList(),
    );
  }
}
class PostList extends StatefulWidget {
  const PostList({super.key});
  @override
  _PostListState createState() => _PostListState();
}
class _PostListState extends State<PostList> {
  late Future<List<Post>> futurePosts;
  @override
  void initState() {
    super.initState();
    futurePosts = fetchPosts();
  }
  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('API Data'),
      ),
      body: FutureBuilder<List<Post>>(
        future: futurePosts,
        builder: (context, snapshot) {
          if (snapshot.connectionState == ConnectionState.waiting) {
            return const Center(child: CircularProgressIndicator());
          } else if (snapshot.hasError) {
            return Center(child: Text('Error: ${snapshot.error}'));
          } else if (snapshot.hasData) {
            return ListView.builder(
              itemCount: snapshot.data!.length,
              itemBuilder: (context, index) {
                return ListTile(
                  title: Text(snapshot.data![index].title),
                  subtitle: Text(snapshot.data![index].body),
                );
              },
            );
          } else {
            return const Center(child: Text('No data available'));
          }
        },
      ),
    );
  }
}
```

Step 5: Add error handling and improve UI

1. Enhance the error handling in your API call:

```
Future<List<Post>> fetchPosts() async {
  try {
    final response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts')).timeout(const Duration(seconds: 10));
    if (response.statusCode == 200) {
      List jsonResponse = json.decode(response.body);
      return jsonResponse.map((post) => Post.fromJson(post)).toList();
    } else {
      throw Exception('Failed to load posts');
    }
  } catch (e) {
    throw Exception('Failed to load posts: $e');
  }
}
```

- The `timeout` method is used to prevent the API call from hanging indefinitely.
- The `try-catch` block handles exceptions that may occur during the request.

2. Update the UI to provide better user feedback:

```
@override
Widget build(BuildContext context) {
  return Scaffold(
    appBar: AppBar(
      title: const Text('API Data'),
    ),
    body: FutureBuilder<List<Post>>(
      future: futurePosts,
      builder: (context, snapshot) {
        if (snapshot.connectionState == ConnectionState.waiting) {
          return const Center(child: CircularProgressIndicator());
        } else if (snapshot.hasError) {
          return Center(
            child: Column(
              mainAxisAlignment: MainAxisAlignment.center,
              children: [
                const Text('Failed to load data.'),
                Text('Error: ${snapshot.error}'),
                ElevatedButton(
                  onPressed: () {
                    setState(() {
                      futurePosts = fetchPosts(); // Retry fetching data
                    });
                  },
                  child: const Text('Retry'),
                ),
              ],
            ),
          );
        } else if (snapshot.hasData) {
          return ListView.builder(
            itemCount: snapshot.data!.length,
            itemBuilder: (context, index) {
              return ListTile(
                title: Text(snapshot.data![index].title),
                subtitle: Text(snapshot.data![index].body),
              );
            },
          );
        } else {
          return const Center(child: Text('No data available'));
        }
      },
    ),
  );
}
```

- Added a retry button to attempt fetching data again in case of an error.

Step 6: Run the final app

If you already have the app running from before, you don’t have to do anything. If you quit that process for some reason, run the script again from the `/home/project` folder.

```
cd /home/project && ./hot_reload.sh
```

Once the app starts, you can refresh the page if the tab is already open, or follow the instructions to launch application on port `8080` as before.

Conclusion and next steps

Congratulations on completing this lab! You have learned how to make API calls in Flutter, handle responses, manage asynchronous operations, and update your app's UI dynamically based on the data fetched.

Author(s)

Skills Network