⟨ Delete data on the internet Make authenticated requests ⟩

Fetch data from the internet

Cookbook › Networking › Fetch data from the internet

Contents

Fetching data from the internet is necessary for most apps. Luckily, Dart and Flutter provide tools, such as the `http` package, for this type of work.

This recipe uses the following steps:

1. Add the `http` package.
2. Make a network request using the `http` package.
3. Convert the response into a custom Dart object.
4. Fetch and display the data with Flutter.

1. Add the `http` package

The `http` package provides the simplest way to fetch data from the internet.

To install the `http` package, add it to the dependencies section of the `pubspec.yaml` file. You can find the latest version of the `http` package the pub.dev.

```
dependencies:
  http: <latest_version>
```

Import the http package.

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

Additionally, in your AndroidManifest.xml file, add the Internet permission.

```
<!-- Required to fetch data from the internet. -->
<uses-permission android:name="android.permission.INTERNET" />
```

2. Make a network request

This recipe covers how to fetch a sample album from the JSONPlaceholder using the `http.get()` method.

```dart
Future<http.Response> fetchAlbum() {
  return http.get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'));
}
```

The `http.get()` method returns a `Future` that contains a `Response`.

- `Future` is a core Dart class for working with async operations. A Future object represents a potential value or error that will be available at some time in the future.
- The `http.Response` class contains the data received from a successful http call.

3. Convert the response into a custom Dart object

While it's easy to make a network request, working with a raw `Future<http.Response>` isn't very convenient. To make your life easier, convert the `http.Response` into a Dart object.

Create an `Album` class

First, create an `Album` class that contains the data from the network request. It includes a factory constructor that creates an `Album` from JSON.

Converting JSON by hand is only one option. For more information, see the full article on JSON and serialization.

```dart
class Album {
  final int userId;
  final int id;
  final String title;

  const Album({
    required this.userId,
    required this.id,
    required this.title,
  });

  factory Album.fromJson(Map<String, dynamic> json) {
    return Album(
      userId: json['userId'],
      id: json['id'],
      title: json['title'],
    );
  }
}
```

Convert the `http.Response` to an `Album`

Now, use the following steps to update the `fetchAlbum()` function to return a `Future<Album>`:

1. Convert the response body into a JSON `Map` with the `dart:convert` package.
2. If the server does return an OK response with a status code of 200, then convert the JSON `Map` into an `Album` using the `fromJson()` factory method.
3. If the server does not return an OK response with a status code of 200, then throw an exception. (Even in the case of a "404 Not Found" server response, throw an exception. Do not return `null`. This is important when examining the data in `snapshot`, as shown below.)

```dart
Future<Album> fetchAlbum() async {
  final response = await http
      .get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'));

  if (response.statusCode == 200) {
    // If the server did return a 200 OK response,
    // then parse the JSON.
    return Album.fromJson(jsonDecode(response.body));
  } else {
    // If the server did not return a 200 OK response,
    // then throw an exception.
    throw Exception('Failed to load album');
  }
}
```

Hooray! Now you've got a function that fetches an album from the internet.

4. Fetch the data

Call the `fetchAlbum()` method in either the `initState()` or `didChangeDependencies()` methods.

The `initState()` method is called exactly once and then never again. If you want to have the option of reloading the API in response to an `InheritedWidget` changing, put the call into the `didChangeDependencies()` method. See `State` for more details.

```dart
class _MyAppState extends State<MyApp> {
  late Future<Album> futureAlbum;

  @override
  void initState() {
    super.initState();
    futureAlbum = fetchAlbum();
  }
  // ···
}
```

This Future is used in the next step.

5. Display the data

To display the data on screen, use the `FutureBuilder` widget. The `FutureBuilder` widget comes with Flutter and makes it easy to work with asynchronous data sources.

You must provide two parameters:

1. The `Future` you want to work with. In this case, the future returned from the `fetchAlbum()` function.
2. A `builder` function that tells Flutter what to render, depending on the state of the `Future`: loading, success, or error.

Note that `snapshot.hasData` only returns `true` when the snapshot contains a non-null data value.

Because `fetchAlbum` can only return non-null values, the function should throw an exception even in the case of a "404 Not Found" server response. Throwing an exception sets the `snapshot.hasError` to `true` which can be used to display an error message.

Otherwise, the spinner will be displayed.

```
FutureBuilder<Album>(
  future: futureAlbum,
  builder: (context, snapshot) {
    if (snapshot.hasData) {
      return Text(snapshot.data!.title);
    } else if (snapshot.hasError) {
      return Text('${snapshot.error}');
    }

    // By default, show a loading spinner.
    return const CircularProgressIndicator();
  },
)
```

Why is fetchAlbum() called in initState()?

Although it's convenient, it's not recommended to put an API call in a `build()` method.

Flutter calls the `build()` method every time it needs to change anything in the view, and this happens surprisingly often. The `fetchAlbum()` method, if placed inside `build()`, is repeatedly called on each rebuild causing the app to slow down.

Storing the `fetchAlbum()` result in a state variable ensures that the `Future` is executed only once and then cached for subsequent rebuilds.

Testing

For information on how to test this functionality, see the following recipes:

- Introduction to unit testing
- Mock dependencies using Mockito

Complete example

```dart
import 'dart:async';
import 'dart:convert';

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

Future<Album> fetchAlbum() async {
  final response = await http
      .get(Uri.parse('https://jsonplaceholder.typicode.com/albums/1'));

  if (response.statusCode == 200) {
    // If the server did return a 200 OK response,
    // then parse the JSON.
    return Album.fromJson(jsonDecode(response.body));
  } else {
    // If the server did not return a 200 OK response,
    // then throw an exception.
    throw Exception('Failed to load album');
  }
}

class Album {
  final int userId;
  final int id;
  final String title;

  const Album({
    required this.userId,
    required this.id,
    required this.title,
  });

  factory Album.fromJson(Map<String, dynamic> json) {
    return Album(
      userId: json['userId'],
      id: json['id'],
      title: json['title'],
    );
  }
}
```



```dart
  State<MyApp> createState() => _MyAppState();
}

class _MyAppState extends State<MyApp> {
  late Future<Album> futureAlbum;

  @override
  void initState() {
    super.initState();
    futureAlbum = fetchAlbum();
  }

  @override
  Widget build(BuildContext context) {
    return MaterialApp(
      title: 'Fetch Data Example',
      theme: ThemeData(
        primarySwatch: Colors.blue,
      ),
      home: Scaffold(
        appBar: AppBar(
          title: const Text('Fetch Data Example'),
        ),
        body: Center(
          child: FutureBuilder<Album>(
```

```
            future: futureAlbum,
            builder: (context, snapshot) {
              if (snapshot.hasData) {
                return Text(snapshot.data!.title);
              } else if (snapshot.hasError) {
                return Text('${snapshot.error}');
              }

              // By default, show a loading spinner.
              return const CircularProgressIndicator();
            },
          ),
        ),
      ),
    );
  }
}
```

⟨ Delete data on the internet Make authenticated requests ⟩