ĐẠI HỌC TÔN ĐỨC THẮNG
TON DUC THANG UNIVERSITY

502071

Cross-Platform Mobile Application Development

Networking

1

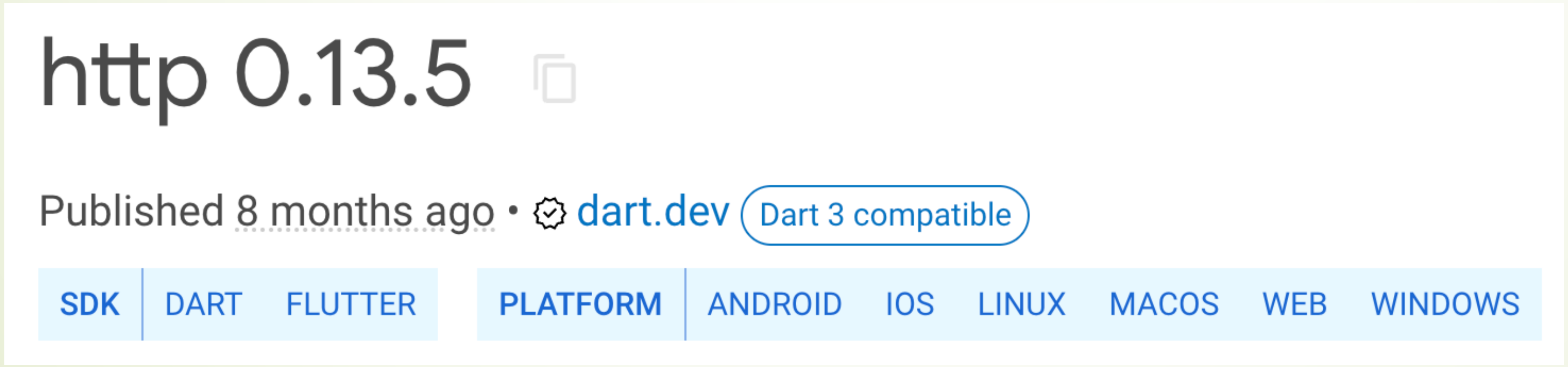
Outline

- http package
- FutureBuilder
- Consume REST API
- Retrofit Package
- Stream
- Web Socket

http package

The http package

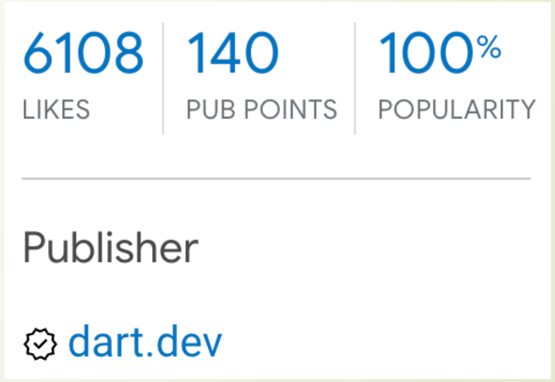
- The http package is a powerful tool in the Flutter framework that allows developers to make HTTP requests to external servers and fetch data from APIs.
- With this package, developers can easily integrate their Flutter applications with web services and create dynamic, data-driven apps.
- This package contains a set of high-level functions and classes that make it easy to consume HTTP resources. It's multi-platform, and supports mobile, desktop, and the browser.



http 0.13.5

Published 8 months ago • dart.dev Dart 3 compatible

SDK DART FLUTTER PLATFORM ANDROID IOS LINUX MACOS WEB WINDOWS



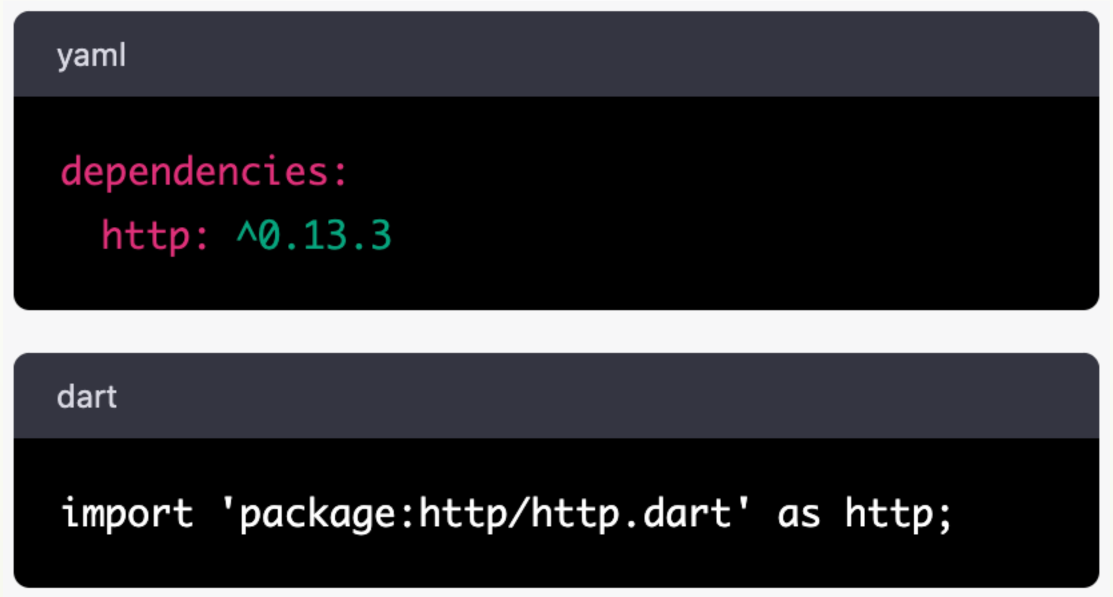
6108 LIKES 140 PUB POINTS 100% POPULARITY

Publisher

dart.dev

The http package

- To use the http package in your project, you need to add it to your pubspec.yaml file and then import it into your Dart code.



yaml

```yaml
dependencies:
  http: ^0.13.3
```

dart

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

The http package

- After importing the package, you can make a HTTP request by creating an instance of the http.Client class and calling one of its methods such as get, post, put, or delete.

```
Future<void> fetchData() async {
    final response = await
        http.get(Uri.parse('https://jsonplaceholder.typicode.com/todos/1'));
    if (response.statusCode == 200) {
        print(response.body);
    } else {
        print('Request failed with status: ${response.statusCode}.');
    }
}
```

The http package

- Once you've made a request and received a response, you may need to parse the data returned by the API. This will depend on the format of the data, which could be JSON, XML, or some other format.

```
import 'dart:convert';

Future<void> fetchData() async {
    final response = await
        http.get(Uri.parse('https://jsonplaceholder.typicode.com/todos/1'));
    if (response.statusCode == 200) {
        final jsonData = jsonDecode(response.body);
        print(jsonData);
    } else {
        print('Request failed with status: ${response.statusCode}.');
    }
}
```

The http package

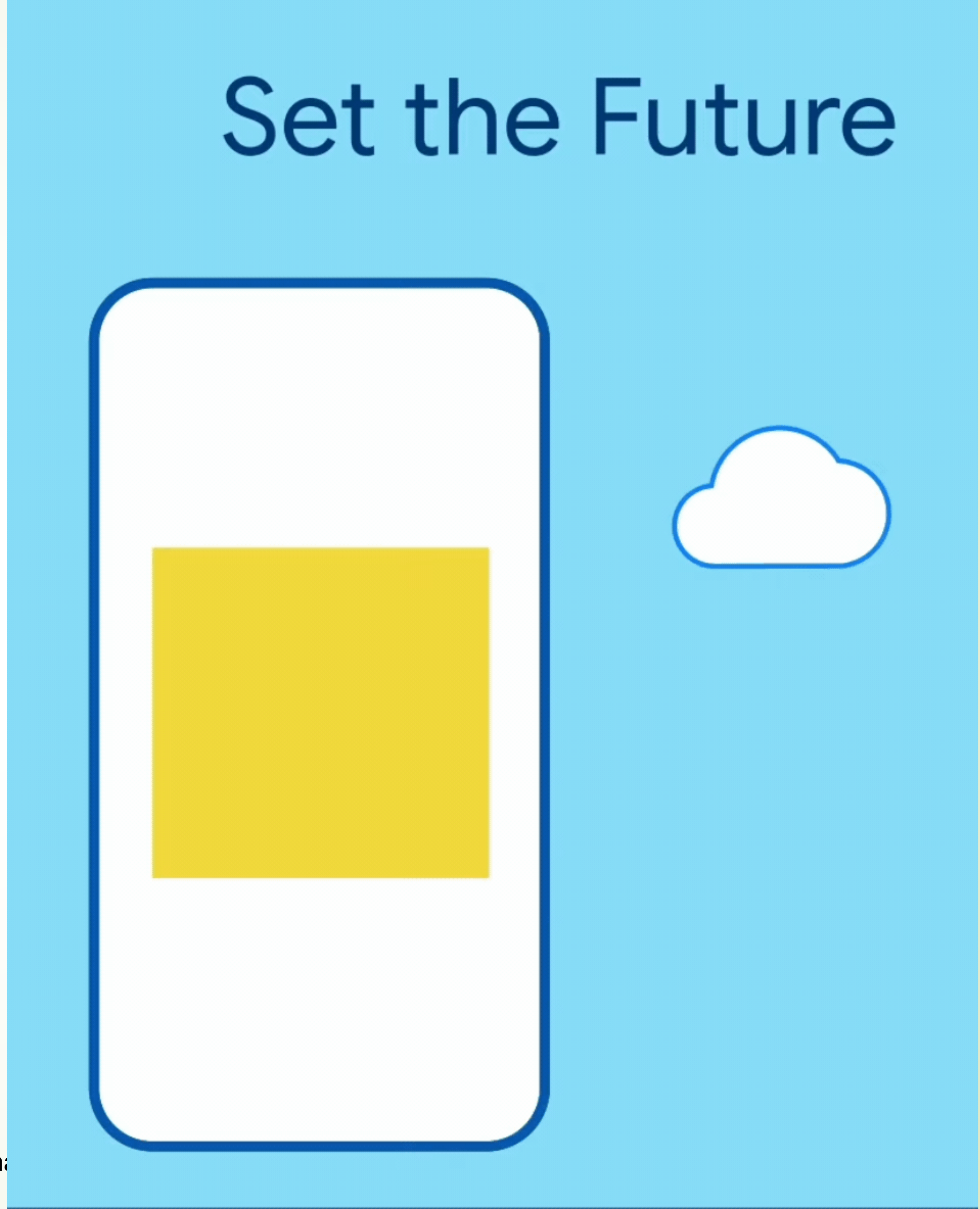
- Finally, it's important to handle errors that may occur during the HTTP request. You can check the status code of the response to determine whether the request was successful or not.

```
Future<void> fetchData() async {
    try {
        final response = await
            http.get(Uri.parse('https://jsonplaceholder.typicode.com/todos/1'));
        // do some processing here
    } catch (error) {
        print('Error occurred: $error’);
        // show error message
    }
}
```

FutureBuilder widget

FutureBuilder

- The FutureBuilder widget allows developers to asynchronously build widgets based on the result of a Future object. This widget provides a seamless way of handling asynchronous data and displaying it to the user, making it an essential tool in developing modern and responsive mobile applications.



When to use the FutureBuilder widget

- **Fetching data from an API**: Use the FutureBuilder widget to display a loading indicator while the data is being fetched, an error message if the data cannot be fetched, and the fetched data once it is available.
- **Processing data in the background**: If you have a time-consuming data processing task that needs to be performed in the background, you can use the FutureBuilder widget is a good choice.
- **Loading data from a database**: When loading data from a local database, you can use the FutureBuilder widget to display a loading indicator while the data is being loaded.
- **Authentication and Authorization**: When handling authentication and authorization in your application, you can use the FutureBuilder widget to display a loading indicator while checking the user's credentials and permissions. You can then display an error message if the credentials are incorrect or if the user does not have the necessary permissions, and the user interface if the user is successfully authenticated and authorized.

The FutureBuilder widget

- **Fetching data from an API**: Use the FutureBuilder widget to display a loading indicator while the data is being fetched, an error message if the data cannot be fetched, and the fetched data once it is available.

How to use FutureBuilder

- **1. Create a Future object that will fetch data from a data source, such as an API or database**

```
Future<String> fetchData() async {
    final response = await
            http.get(Uri.parse('https://jsonplaceholder.typicode.com/todos/1'));
    if (response.statusCode == 200) {
        return response.body;
    } else {
        throw Exception('Failed to load data');
    }
}
```

How to use FutureBuilder

- **2. In your widget tree, add a FutureBuilder widget and pass the future object to its future parameter.**

```
FutureBuilder<String>(
    future: fetchData(),
    builder: (BuildContext context, AsyncSnapshot<String> snapshot) {
        // TODO: return a widget based on the snapshot's state
    },
),
```

- The BuildContext parameter is useful when building widgets that depend on context-sensitive information, such as theme data or localization data. The BuildContext can also be used to create new widgets or to access inherited widgets that are higher up in the widget tree.

How to use FutureBuilder

- **2. In your widget tree, add a FutureBuilder widget and pass the future object to its future parameter.**

```
FutureBuilder<String>(
    future: fetchData(),
    builder: (BuildContext context, AsyncSnapshot<String> snapshot) {
        // TODO: return a widget based on the snapshot's state
    },
),
```

- The AsyncSnapshot parameter is used to provide information about the state of the asynchronous operation that the FutureBuilder is waiting on. The AsyncSnapshot object has three main properties: data, error, connectionState.

How to use FutureBuilder

- **3. In the builder parameter, return a widget based on the snapshot's state.**

```
FutureBuilder<String>(
  future: fetchData(),
  builder: (BuildContext context, AsyncSnapshot<String> snapshot) {
    if (snapshot.connectionState == ConnectionState.done) {
      if (snapshot.hasData) {
        return Text(snapshot.data!);
      } else if (snapshot.hasError) {
        return Text('Error: ${snapshot.error}');
      }
    }
    return CircularProgressIndicator();
```

Connection State

- The connectionState property of an AsyncSnapshot object indicates the current state of the future, while the hasData and hasError properties indicate whether the future completed with data or an error, respectively.
- List of connection states:
  - ConnectionState.none: This is the initial state before a future has begun executing.
  - ConnectionState.waiting: This state indicates that a future is currently executing and has not yet completed.
  - ConnectionState.active: This state is similar to waiting, but is used specifically for futures that involve a stream or other continuous source of data.
  - ConnectionState.done: This state indicates that a future has completed, either with data or an error.

Benefits of using FutureBuilder

- **Simplifies asynchronous programming**: The FutureBuilder widget provides a convenient way to display loading indicators, error messages, and data based on the state of a Future object.
- **Increases app performance**: By handling asynchronous data asynchronously, the FutureBuilder widget allows the UI to remain responsive while data is being fetched or processed in the background.
- **Improves user experience**: The FutureBuilder widget allows the user to know when the app is loading data, when an error occurs, or when data is available to view.
- **Flexible customization**: Developers can define their own loading indicators, error messages, and widgets to display data based on the state of the Future object.

FutureBuilder Best Practices

- Make sure the future object passed to the FutureBuilder widget is created outside of the build method.
- Handle errors appropriately in the builder function. If the future object completes with an error, make sure to display an error message or take other appropriate action.
- If the future object depends on external data or user input, consider using a debounce or throttle mechanism to avoid excessive requests or updates.
- Avoid nesting multiple FutureBuilder widgets within each other. Instead, consider using a single FutureBuilder widget that returns a complex widget tree based on the state of the future.
- Consider using other asynchronous Flutter widgets, such as StreamBuilder or ValueListenableBuilder, if they better suit your use case. FutureBuilder is best used for one-time asynchronous operations that have a clear start and end.

Consume REST API

Using REST Api

- REST API enables developers to build mobile applications that can easily communicate with the server-side backend, regardless of the technology stack used in the backend.
- Additionally, REST API allows developers to integrate with multiple data sources such as databases, file systems, and cloud services, providing a more comprehensive and versatile data source for the application.
- One of the most important features of any mobile application is the ability to communicate with a server or a database. To achieve this, Flutter provides a package called 'http', which enables developers to make HTTP requests to REST APIs.

REST Api vs Local Storage

- If the application requires data that changes frequently or is unique to each user, such as user-generated content or real-time data, then REST API is a better choice. On the other hand, if the data is relatively static and does not change frequently, local storage may be a better option, especially if the application needs to work offline.
- If the data is complex and requires complex querying and filtering, a REST API may be a better option, as it can handle the processing on the server-side. However, if the data is simple and straightforward, local storage may be a more straightforward and efficient solution.

Define the model

- To work with the product resource, we need to create a Product model. Create a new file named product_model.dart in the lib/models directory and add the following code.

```
class Product {
    int id;
    String name;
    String description;
    double price;

    Product({
        required this.id,
        required this.name,
        required this.description,
        required this.price});
}
```

```
factory Product.fromJson(Map<String, dynamic> json) =>
    Product(
        id: json['id'],
        name: json['name'],
        description: json['description'],
        price: json['price'],
    );

    Map<String, dynamic> toJson() => {
        'id': id,
        'name': name,
        'description': description,
        'price': price,
    };
```

Fetching data from the internet

- To retrieve all products, create a new file named product_service.dart in the lib/services directory and add the following code:

```
Future<List<Product>> getProducts() async {
    try {
        final response = await http.get(Uri.parse('http://abc.com/products'));
        if (response.statusCode == 200) {
        final parsed = jsonDecode(response.body) as Map<dynamic>;
        return parsed.map<Product>((json) => Product.fromJson(json)).toList();
    } else {
        throw Exception('Failed to load products');
    }
    } catch (e) {
        throw Exception('Failed to load products');
    }
}
```

Send data to the internet

- To create a new product, we send a http post request to the api endpoint, along with the required information.

```
Future<Product> createProduct(Product product) async {
    final response = await http.post(
        Uri.parse('http://abc.com/products'),
        headers: {
            'Content-Type': 'application/json; charset=UTF-8',
        },
        body: jsonEncode(product.toJson()));

    if (response.statusCode == 201) {
        return Product.fromJson(jsonDecode(response.body));
    } else {
        throw Exception('Failed to create product');
    }
}
```

Send data to the internet

- To fetch data from most web services, you need to provide authorization. There are many ways to do this, but perhaps the most common uses the Authorization HTTP header.
- The http package provides a convenient way to add headers to your requests. Alternatively, use the HttpHeaders class from the dart:io library.

```
Future<Product> createProduct(Product product) async {
    final response = await http.post(Uri.parse('http://abc.com/products/a123'),
        headers: <String, String>{
            'Content-Type': 'application/json; charset=UTF-8',
            'Authorization': 'Bearer your_api_token_here',
        },
        body: jsonEncode(product.toJson()));

    // handle respose here
}
```

Update data over the internet

- To update a product, we use the http.put() method to send the request with the updated product data encoded as JSON in the request body.

```
Future<Product> updateProduct(Product product) async {
    final response = await http.put(Uri.parse('http://abc.com/products/a123'),
        headers: {
            'Content-Type': 'application/json; charset=UTF-8’,
        },
        body: jsonEncode(product.toJson()));

    if (response.statusCode == 200) {
        return Product.fromJson(jsonDecode(response.body));
    } else {
        throw Exception('Failed to update product');
    }
}
```

Delete data on the internet

- To delete a product, we use the http.delete() method to send the request with the product id in the path parameter.

```
Future<void> deleteProduct(int id) async {
    final response = await http.delete(Uri.parse('http://abc.com/products/a123'));
    if (response.statusCode != 200) {
        throw Exception('Failed to delete product');
    }
}
```

Decoding json in the background

- By default, Dart apps do all of their work on a single thread. If you perform an expensive computation more than 16 milliseconds, your users experience jank.
- Decoding JSON can be a time-consuming operation, it's best to perform it in the background to avoid blocking the UI thread and keeping the app responsive.
- One way to decode JSON in the background is to use a compute() function provided by Flutter. The compute() function takes a function as an argument that performs the JSON decoding operation and returns the decoded data. It then runs the function in a separate isolate (a lightweight thread) and returns the result to the main UI thread once the operation is completed.

Decoding json in the background

```
// this function must have the global scope
List<Product> _decodeProducts(List<dynamic> jsonList) {
    return jsonList.map((json) => Product.fromJson(json)).toList();
}

Future<List<Product>> getProducts() async {
    final response = await http.get(Uri.parse('http://abc.com/products'));
    // assume that the response was successful
    final jsonList = json.decode(response.body) as List<dynamic>;
    return await compute(_decodeProducts, jsonList);
}
```

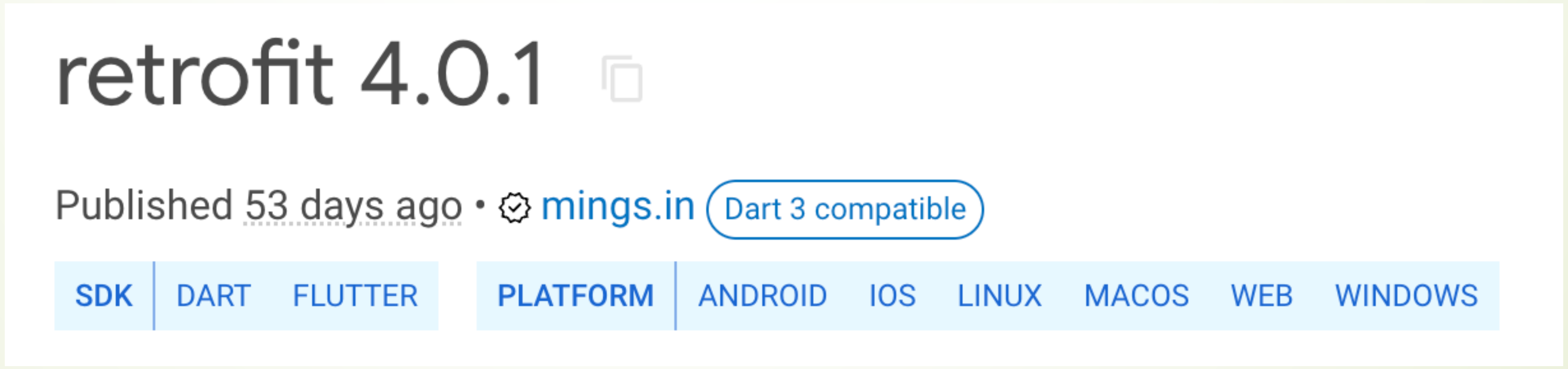
Best Practices

- **Use a separate service class**: Create a separate class for each REST API resource that handles all the HTTP requests and responses. This will keep your code organized and easy to maintain.
- **Use asynchronous programming**: Use asynchronous programming with async and await to avoid blocking the user interface (UI) thread. This will ensure that the app remains responsive and doesn't freeze while waiting for responses from the server.
- **Error handling**: Always handle errors and exceptions properly in your code. This means checking the response status codes, catching exceptions, and displaying error messages to the user.
- **Security**: Ensure that you are using secure communication protocols like HTTPS to protect your app and user data from security threats. You should also ensure that you are validating all user input on the client-side to prevent security vulnerabilities like injection attacks.
- **Testing**: Always test your API integration code thoroughly before releasing it to production. This will help to catch any bugs or issues early on and ensure that your app is functioning correctly.

Retrofit Packge

Retrofit package

- Retrofit is a popular package for making HTTP requests and handling responses. It's a type-safe HTTP client that makes it easy to consume RESTful APIs and parse JSON data into Dart objects.
- Retrofit is inspired by the Retrofit library for Android and is designed to work seamlessly with Flutter. Retrofit provides an elegant and concise way to define API endpoints, request parameters, and response types using annotations and interfaces. This makes it simple to maintain and modify code over time, especially when working with large and complex APIs.


retrofit 4.0.1

Published 53 days ago • mings.in Dart 3 compatible

SDK DART FLUTTER PLATFORM ANDROID IOS LINUX MACOS WEB WINDOWS

Retrofit package

- Add the retrofit package to your Flutter project by adding the following line to your pubspec.yaml

```
dependencies:
    retrofit: ^4.0.1
    json_annotation: ^4.8.0
    dio: ^5.1.1

dev_dependencies:
    build_runner: ^2.3.3
    retrofit_generator: ^6.0.0+1
```

Retrofit package

- **dio** is a popular HTTP client library for Dart and Flutter that is often used in conjunction with retrofit. It provides a convenient way to make HTTP requests and handle responses, with features such as request cancellation, interceptors, and progress monitoring.
- **json_annotation** is a package used to generate code for JSON serialization and deserialization. It provides annotations that can be used to specify how Dart objects should be converted to JSON and vice versa. Retrofit uses this package to automatically parse JSON response data into Dart objects.
- **retrofit_generator** is a code generator that generates the implementation of the REST API interface annotated with the **@RestApi** annotation. The generated code includes all the boilerplate code required to make the HTTP requests, handle the response, and convert the response data into Dart objects.
- **build_runner** is a build tool used to run code generators like **retrofit_generator** and **json_annotation**. It automates the process of generating code, making it easier to use Retrofit with minimal manual configuration.

Define API Service Interface

- Define the API service interface which contains the methods for making HTTP requests.

```
// this code is placed in the file 'api_service.dart'
import 'package:retrofit/retrofit.dart';
import 'package:dio/dio.dart';
part 'api_service.g.dart';

@RestApi(baseUrl: "https://example.com/api/")
abstract class ApiService {
     factory ApiService(Dio dio) = _ApiService;
     // define the CRUD methods here
}
```

- We create an abstract class called **ApiService** and annotated it with **@RestApi** to indicate that it is a Retrofit API service. The **baseUrl** parameter indicates the base URL for the API.
- We also defin a **factory constructor** that takes a **Dio** object as an argument, which is required for the Retrofit library to make HTTP requests.

Define API Service Methods

- The ApiService class in Retrofit only contains method signatures. This is because Retrofit is a library that uses Dart's reflection capabilities to generate the method implementations at runtime, based on the annotations and parameter types defined in the ApiService interface.

```
@RestApi(baseUrl: "https://example.com/api/")
abstract class ApiService {
    factory ApiService(Dio dio) = _ApiService;
    @GET("/users")
    Future<List<User>> getUsers();
}
```

- The @GET annotation is used to indicate that this method makes a GET request to the specified URL, which is "/users" in this case. We also specify the expected return type of the method (e.g., Future<User>).

Define API Service Methods

- The @POST annotation is used to indicate that this method makes a POST request to the specified URL, which is "/users" in this case to create a new user with json data in the request body.

```
@RestApi(baseUrl: "https://example.com/api/")
abstract class ApiService {
    factory ApiService(Dio dio) = _ApiService;

    @POST("/users")
    Future<User> createUser(@Body() User user);
}
```

- @Path: This annotation is used to substitute a value for a path parameter in the URL. For example, @GET("/users/{id}") would specify an endpoint that expects a id path parameter.
- @Query: This annotation is used to add query parameters to the URL. For example, @GET("/users") with @Query("page") int page would add a page query parameter to the URL.
- @Body: This annotation is used to specify the request body for a POST, PUT, or PATCH request. The parameter annotated with @Body is serialized to JSON and sent as the request body.

Define API Service Methods

```
abstract class ApiService {
    factory ApiService(Dio dio) = _ApiService;

    @GET("/users")
    Future<List<User>> getUsers();

    @GET("/users/{id}")
    Future<User> getUserById(@Path("id") int id);

    @POST("/users")
    Future<User> createUser(@Body() User user);

    @PUT("/users/{id}")
    Future<User> updateUserById(@Path("id") int id, @Body()
    User user);

    @DELETE("/users/{id}")
    Future<void> deleteUserById(@Path("id") int id);
}
```

Define Resource Data Type

- We have also defined a User class to represent a user. This class has an id, name, and email field, as well as toJson and fromJson methods for converting between JSON and User objects.

```
@JsonSerializable()
class User {
    final int id;
    final String name;
    final String email;

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

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

    Map<String, dynamic> toJson() => {...};
}
```

Define API Service Interface

- When you first create the ApiService class as shown in the previous example and try to run your app, you may encounter an error message.

```
@RestApi(baseUrl: "https://example.com/api/")
abstract class ApiService {
    factory ApiService(Dio dio) = _ApiService;
    // define the CRUD methods here
}
```

- This error occurs because the ApiService class is an abstract class and we have not provided any implementations for the methods in the class. However, we have annotated the ApiService class with @RestApi and specified a baseUrl, indicating that we intend to use this class as a Retrofit API service.
- To fix this error, we need to generate the implementation of the ApiService class using the build_runner package. This package reads the @RestApi annotation and generates the required implementation code for us.

API Service Implementation

- To generate the implementation code, run the following command in the terminal:

```
dart run build_runner build

flutter pub run build_runner build
```

- This command will generate a file named api_service.g.dart in the same directory as the api_service.dart file. This file contains the generated implementation of the ApiService class, including the missing method implementations.
- Once the implementation code has been generated, you can import the generated file in your Dart code like this:

```
import 'package:my_app/api_service.g.dart';
```

- Now you can create an instance of the ApiService class and start using its methods to make API requests.

Consume the API

- Then, you could use the methods from the ApiService class to make HTTP requests and interact with the user API. For example, here's how you might use the createUser method to create a new user:

```
final apiService = ApiService(Dio());

final newUser = User(id: 1, name: 'John Doe', email: 'johndoe@example.com');
final createdUser = await apiService.createUser(newUser);
```

- This would create a new user with the ID of 1, the name "John Doe", and the email "johndoe@example.com". The createdUser variable would contain the newly created user object returned from the API.

Stream

Introduction to Stream

- In programming, a stream is a sequence of data that can be asynchronously produced and consumed. Think of it like a pipeline that carries data from one point to another. The data can flow through the pipeline at any time, and the pipeline can be connected to any number of sources and destinations.
- In Dart and Flutter, a stream is an abstraction that represents a sequence of asynchronous events. It's a fundamental concept that's used throughout the Dart language and Flutter framework.

Introduction to Stream

- A stream in Flutter can be thought of as a source of asynchronous events that can be consumed by widgets and other components in the app. For example, you might use a stream to receive real-time updates from a database or web socket server, or to listen for user input events.
- To work with a stream in Flutter, you typically use the Stream and StreamController classes from the dart:async library. A StreamController is an object that can produce events on a stream, while a Stream is an object that represents a sequence of events.
- You can listen for events on a stream using the Stream.listen() method, which registers a callback function that is called every time an event is emitted on the stream.

Create a Stream Controller

- You can create a stream controller using the StreamController class. This class provides you with a way to add data to the stream and also listen to data from the stream.

```
final StreamController<String> _streamController =
                                StreamController<String>();
```

- To add data to the stream, you can use the add method provided by the stream controller.

```
_streamController.add("Hello World!");
```

Listen to data from the stream

- To listen to data from a stream in, you can use the Stream.listen() method to register a callback function that is called every time a new event is emitted on the stream.

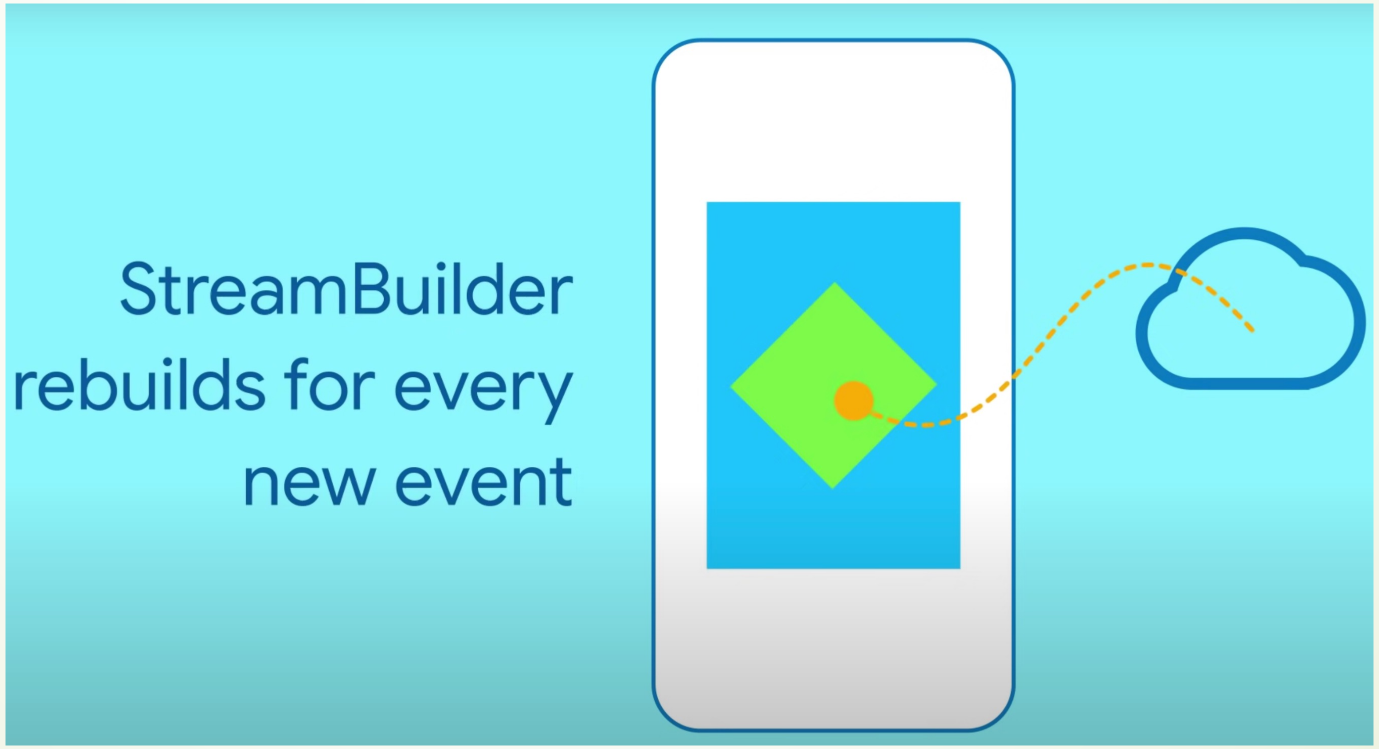
```
Stream<int> _counterStream = Stream<int>.
                              periodic(Duration(seconds: 1), (count) => count).take(10);

        @override
        void initState() {
            super.initState();
            _counterStream.listen((count) {
                print('Received count: $count');
            });
        }
```

- It's important to properly dispose of stream subscriptions when they are no longer needed. This can be done using the StreamSubscription.cancel() method.

StreamBuilder widget

StreamBuilder widget

- The StreamBuilder widget in Flutter is a powerful tool for working with asynchronous data streams. It allows you to easily build reactive user interfaces that automatically update in response to changes in a stream.



StreamBuilder widget

- The StreamBuilder widget takes a Stream as input and a builder function that returns a widget tree. The builder function is called every time a new event is emitted on the stream, and it is passed a snapshot object that contains the latest data from the stream. You can use this snapshot object to update the widget tree in response to changes in the stream.

```
StreamBuilder(
    stream: myStream,
    builder: (BuildContext context, AsyncSnapshot snapshot) {
        // Build widget tree based on snapshot data
    },
)
```

Build widget tree based on the snapshot data

- In the builder function, you can use the AsyncSnapshot object to access the latest data from the stream. The AsyncSnapshot object has a data property that contains the latest data emitted by the stream, as well as other properties like hasData, hasError, and connectionState that can be used to handle different states of the stream.

```
StreamBuilder(
    stream: myStream,
    builder: (BuildContext context, AsyncSnapshot<int> snapshot) {
        if (snapshot.hasData) {
            return Text('Count: ${snapshot.data}');
        } else {
            return Text('Waiting for data...');
        }
    },
)
```

Handle errors and connection states

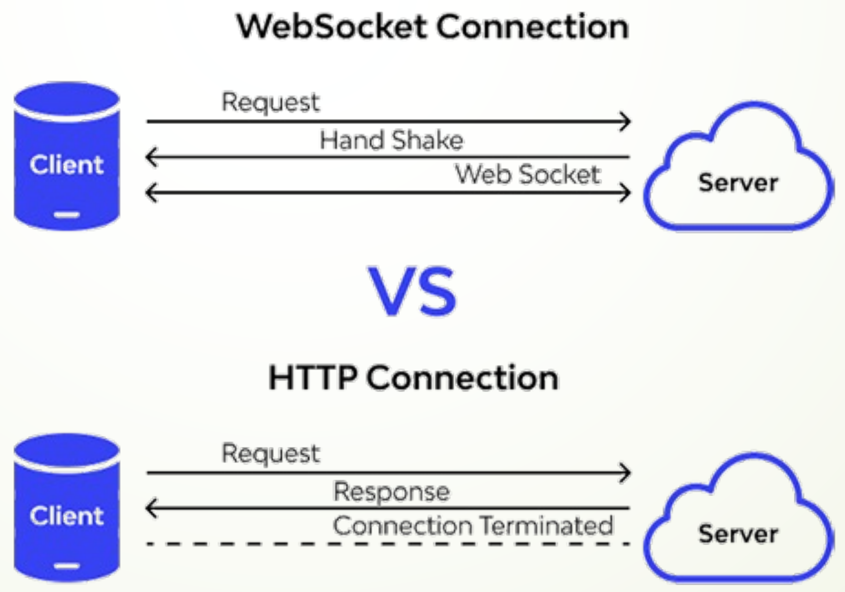
- In addition to handling data, you can also use the AsyncSnapshot object to handle errors and connection states of the stream. For example, you might display an error message if the stream emits an error, or display a loading spinner if the connection state is waiting.

```
StreamBuilder(
    stream: myStream,
    builder: (BuildContext context, AsyncSnapshot<int> snapshot) {
        if (snapshot.hasData) {
            return Text('Count: ${snapshot.data}');
        } else if (snapshot.hasError) {
            return Text('Error: ${snapshot.error}');
        } else if (snapshot.connectionState == ConnectionState.waiting) {
            return CircularProgressIndicator();
        } else {
            return Text('Waiting for data...');
        }
    },
)
```

Web Socket

Web Socket

- WebSockets are a technology that enables two-way communication between a client and a server over a single, long-lived connection. They provide a persistent connection that allows for real-time data transfer and can be used for a variety of applications, such as chat applications, online gaming, and stock market tracking. In Flutter, the dart:io library provides a built-in WebSocket class that allows developers to establish and manage WebSocket connections.



Using WebSocket in Flutter

- To use the WebSocket class in Flutter, we need to first import the dart:io library. We can do this by adding the following import statement at the top of our Dart file:

```
import 'dart:io';
```

- To establish a WebSocket connection, we need to create an instance of the WebSocket class and pass in the URL of the WebSocket server as a parameter. We can do this using the WebSocket.connect() method, which returns a Future that resolves to a WebSocket instance once the connection is established.

```
final socket = await WebSocket.connect('ws://localhost:8080');
```

Listen for events

- Once we have established a WebSocket connection, we can listen for events on the socket by calling the WebSocket.listen() method.

```
socket.listen((message) {
    print('Received message: $message');
});
```

- To send messages over the WebSocket connection, we can use the WebSocket.add() method. This method takes a message as a parameter and sends it to the WebSocket server.

```
socket.add('Hello, server!');
```

- When we are finished using the WebSocket connection, we should close it to free up system resources. We can do this using the WebSocket.close() method.

```
socket.close();
```

Send a binary file

- Before we can send the file data over the WebSocket, we need to convert it to a format that can be sent over the network. One common way to do this is to convert the file data to a byte array.

```
final socket = await
WebSocket.connect('ws://localhost:8080');
final file = File('path/to/file');
final bytes = await file.readAsBytes();
socket.add(bytes);
```

- On the receiving end of the WebSocket connection, you would simply listen for messages and handle them as byte arrays:

```
socket.listen((message) {
    // Handle the message as a byte array
});
```

The web_socket_channel package

- The web_socket_channel package provides the tools you need to connect to a WebSocket server.
- A few reasons why you might want to use the WebSocketChannel package instead of the WebSocket class:
  - **Higher-level API**: The WebSocketChannel package provides a higher-level API than the WebSocket class. It abstracts away some of the low-level details of the WebSocket protocol and provides a more user-friendly interface for sending and receiving messages over a WebSocket connection.
  - **Interoperability**: The WebSocketChannel package is designed to work with other packages that use the StreamChannel API, such as http and grpc. This makes it easier to integrate WebSocket communication with other networking protocols in your Dart application.
  - **Testing**: The WebSocketChannel package provides a mock implementation of the WebSocket protocol that can be used for testing. This makes it easier to write unit tests for WebSocket communication in your Dart application.

The web_socket_channel package

- In Flutter, use the following line to create a WebSocketChannel that connects to a server:

```
final channel = WebSocketChannel.connect(
    Uri.parse('ws://localhost:8080'),
);
```

- The WebSocketChannel instance provides a Stream for incoming messages and a StreamSink for outgoing messages.

```
channel.stream.listen((message) {
    print('Received message: $message');
});

// Send outgoing messages
channel.sink.add('Hello WebSocket server!');
```