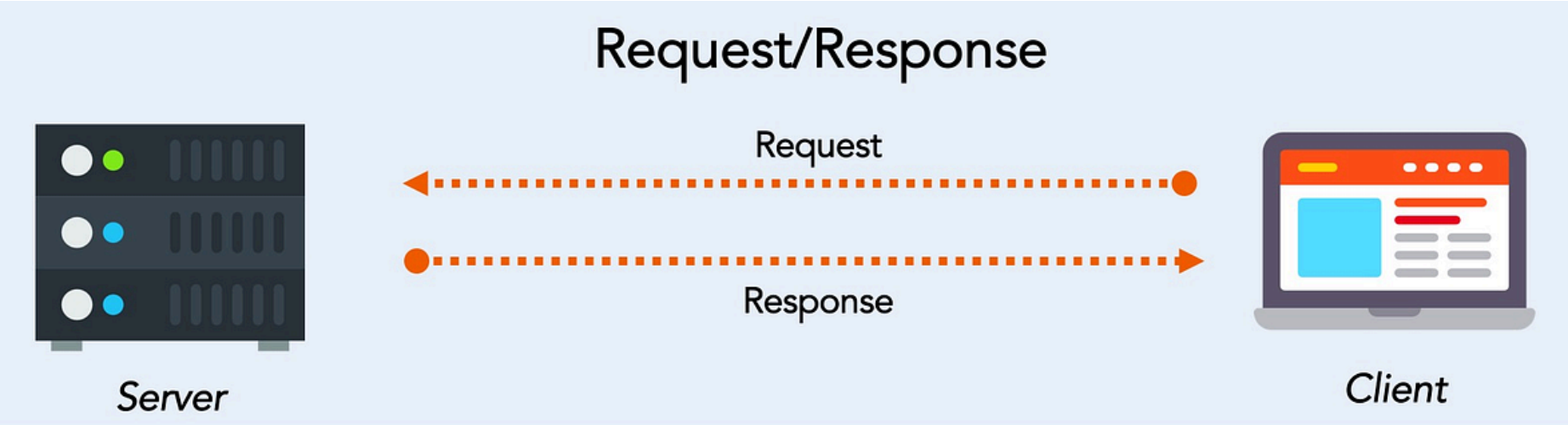
Day 7:
Networking in Flutter

Networking - Overview

- In **mobile application** development, there are two main components:
  - The **client** (the mobile device running your app)
  - The **server** (a remote machine or cloud service)
- These components communicate with each other over a **network**.

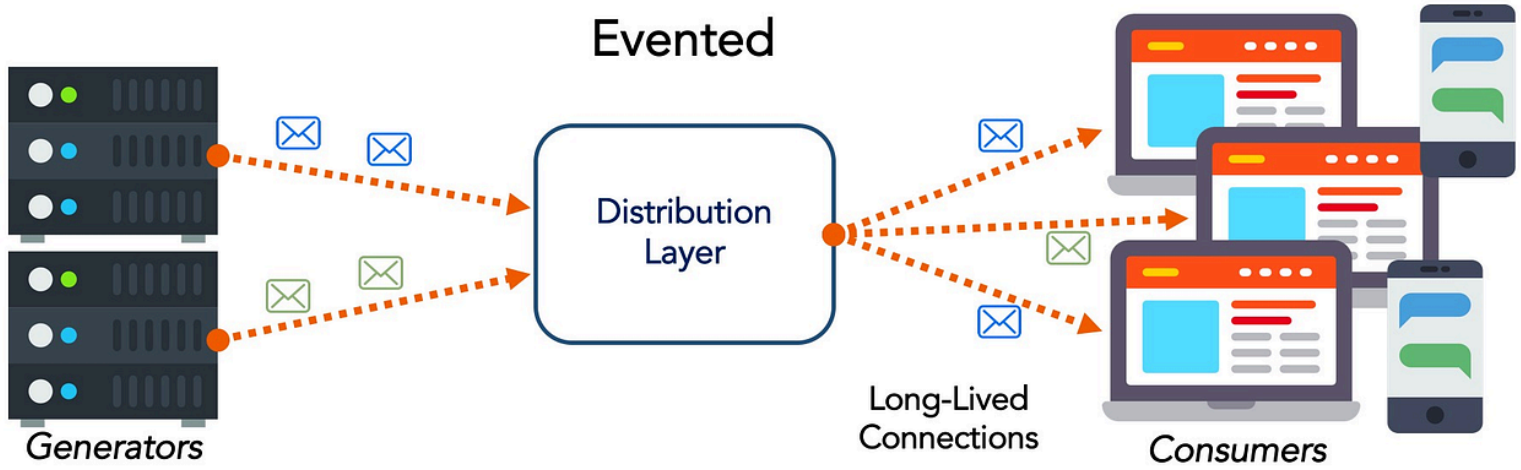
Networking - Events

- **Networking** in mobile applications refers to the process of sending **requests** from the client (your app) to a server and handling the **responses** received.
  - **Request:** The client sends a request to the server to access resources or perform actions.
  - **Response:** The server processes the request and sends back a response containing the requested data or the result of the action.



Networking - Distributed Systems

- Mobile applications often interact with **distributed systems**, where multiple servers work together to provide services to users.
- Key concepts:
  - **Load Balancing:** Distributes requests across servers for better performance.
  - **Scalability:** Adds resources to handle more users or data.



Networking - HTTP Requests

In Flutter, you can perform HTTP requests using packages like `http` and `dio`.

- `http` package: Simple and easy to use for basic HTTP requests.
- `dio` package: Offers advanced features like interceptors, global configuration, and file downloading/uploading.

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

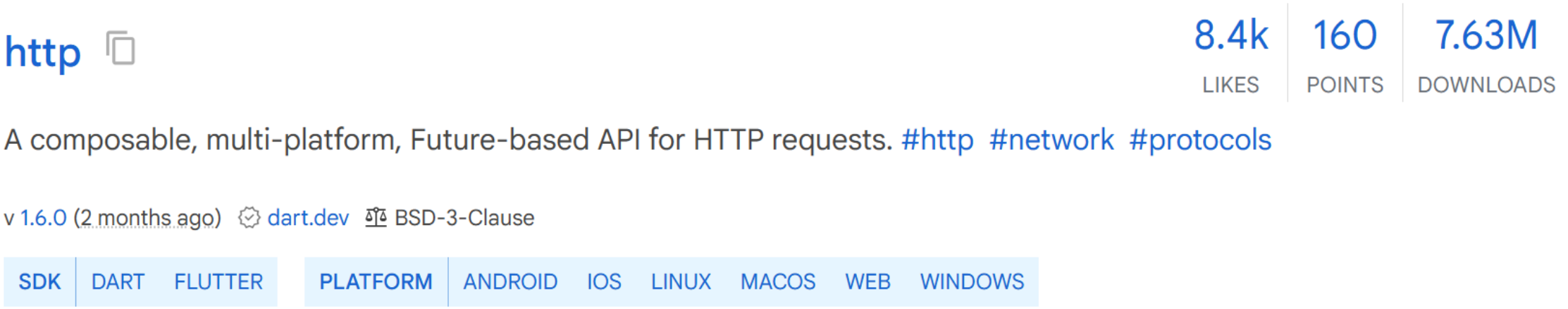
```dart
import 'package:dio/dio.dart';
```

HTTP - Overview

The **http package** is a popular choice for making HTTP requests in Flutter. It provides a simple way to send GET, POST, PUT, and DELETE requests and handle responses.

- To use the `http` package, add it to your `pubspec.yaml`:

```
dependencies:
  http: ^0.13.4
```



HTTP - GET/POST Requests

GET Request

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

POST Request

```
final response = await http.post(
    Uri.parse("https://api.example.com/data"),
    body: {"key": "value"},
);
```

HTTP - Handling Async Data in Flutter using Future

WRONG – Inside `build()`:

```
Widget build(BuildContext context) {
    final futureItem = widget._request.execute(); // BAD!
}
```

- `widget._request.execute()` is called every time `build()` runs, causing multiple requests.

RIGHT – Inside `initState()`:

```
late final Future<Item> futureItem;
@override
void initState() {
    super.initState();
    futureItem = widget._request.execute(); // Called once
}
```

- `futureItem` is initialized once when the widget is created.

HTTP - Handling Async Data with FutureBuilder

```
FutureBuilder<Item>(
    future: futureItem,
    builder: (context, snapshot) {
        if (snapshot.connectionState == ConnectionState.waiting) {
            return CircularProgressIndicator();
        } else if (snapshot.hasError) {
            return Text("Error: ${snapshot.error}");
        } else if (snapshot.hasData) {
            final item = snapshot.data!;
            return Text("Item: ${item.name}");
        } else {
            return Text("No data");
        }
    },
);
```

- `AsyncSnapshot` properties: `hasError`, `hasData`, `data`

HTTP - POST Requests & Headers

POST with body:

```
final response = await http.post(
    url,
    body: "data",
    headers: {"Authorization": "api_key"},
);
```

- **Body types:** `String`, `List<int>`, `Map<String, String>`
- **Headers:** Pass as `Map<String, String>`

HTTP - Multiple Requests

```
final client = http.Client();
try {
    final r1 = await client.get(url1);
    final r2 = await client.get(url2);
} finally {
    client.close();
}
```

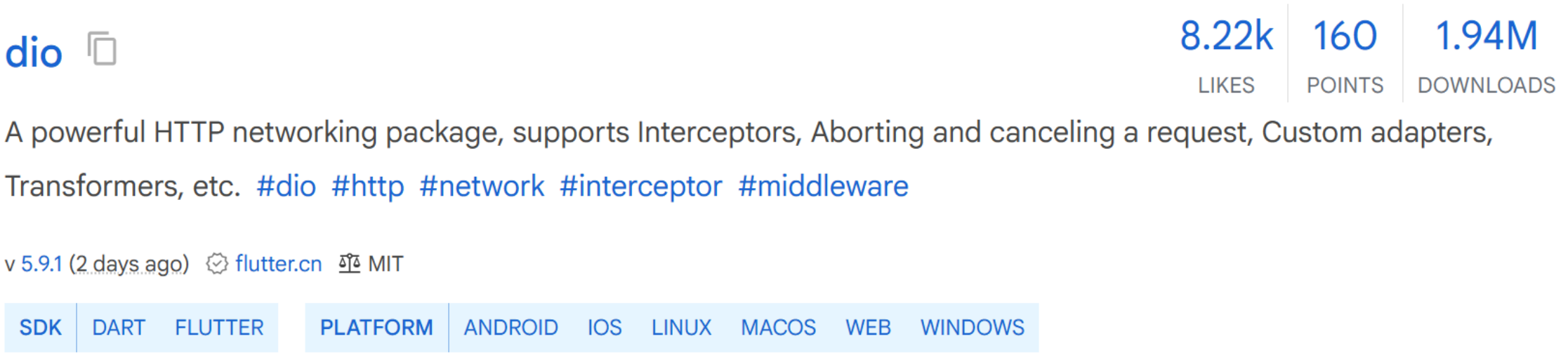
- More efficient than separate `http.get()` calls.

Dio - Overview

`dio` is a powerful HTTP client for Dart/Flutter with advanced features. It supports interceptors, global configuration, form data, file downloading/uploading, and more.

- To use `dio`, add it to your `pubspec.yaml`:

```
dependencies:
  dio: ^5.0.0
```



Dio - GET/POST Requests

GET Request

```
final dio = Dio();
final response = await dio.get<String>("https://api.example.com/data");
```

POST Request

```
final response = await dio.post<String>(
    "https://api.example.com/data",
    data: {"key": "value"},
);
```

Dio – Multiple Requests

```
final dio = Dio(BaseOptions(
    baseUrl: "https://api.example.com/",
    connectTimeout: 3000,
));

final responses = await Future.wait([
    dio.get("/version"),
    dio.get("/products"),
    dio.post("/login", data: {...}),
]);
```

Dio – Downloading Files with Progress

```
class DownloadProgress with ChangeNotifier {
    double _progress = 0.0;
    double get progress => _progress;

    void start({required String url, required String filename}) async {
        await dio().download(url, path,
            onReceiveProgress: (received, total) {
                _progress = received / total * 100;
                notifyListeners();
            },
        );
    }
}
```

Dio - Using Download Progress in UI

```
Consumer<DownloadProgress>(
    builder: (context, status, _) {
        return RaisedButton(
            child: Text("${status.progress.toStringAsFixed(1)} %"),
            onPressed: () => status.start(url: url, filename: "file.pdf"),
        );
    },
);
```

Dio – Uploading Files with Progress

FormData for multipart upload:

```
final formData = FormData.fromMap({
    "nickname": "Roberto",
    "file": await MultipartFile.fromFile(filePath),
});

await dio.post<String>("/upload", data: formData);
```

- Track upload progress with `onSendProgress`.

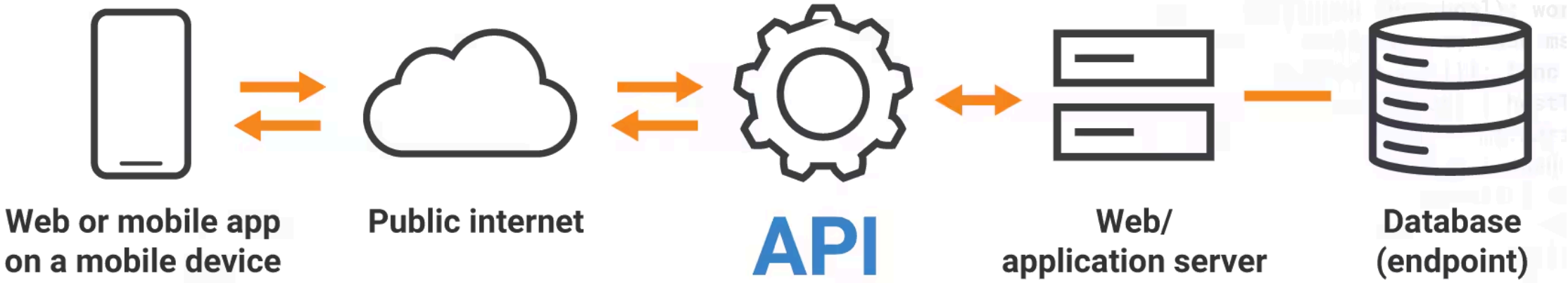
Dio – Upload Progress Model

```
class UploadProgress with ChangeNotifier {
    double _progress = 0.0;
    double get progress => _progress;

    void start({required String url, required String filename}) async {
        await dio().post(url, data: formData,
            onSendProgress: (sent, total) {
                _progress = sent / total * 100;
                notifyListeners();
            },
        );
    }
}
```

API - Overview

- **APIs** (Application Programming Interfaces) are sets of rules that allow different software applications to communicate with each other.



API - Types

- **REST API:** Uses HTTP methods and is stateless; commonly used for web and mobile applications.
- **SOAP API:** Uses XML messaging and is protocol-based; often used in enterprise environments.
- **GraphQL API:** Allows clients to request exactly the data they need, using a single endpoint.
- **gRPC API:** Uses protocol buffers and HTTP/2; suitable for high-performance, real-time communication.
- **WebSocket API:** Enables two-way, real-time communication between client and server.

TIP

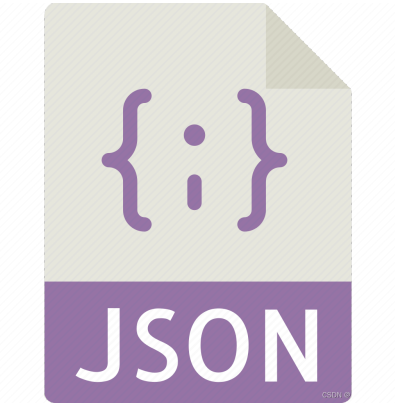
OpenAPI/Swagger is a specification for building APIs, providing a standard way to describe RESTful APIs.

Working with JSON in Flutter

JSON - Overview

JSON (JavaScript Object Notation) is a lightweight data interchange format that is easy for humans to read and write, and easy for machines to parse and generate.

- Commonly used in web and mobile applications for data exchange between client and server.



JSON - Structure

- **Arrays:** Ordered lists enclosed in `[ ]`
- **Objects:** Key-value pairs enclosed in `{ }`
- **Datatypes:** Strings, numbers, booleans, arrays, objects, or `null`

Example:

```
{
  "id": 1,
  "name": "Alberto",
  "isActive": true,
  "tags": ["developer", "flutter"],
  "address": {
    "street": null,
    "city": "Phnom Penh"
  }
}
```

JSON - Parsing in Dart/Flutter

- Import the `dart:convert` library to work with JSON in Dart.
- Use `jsonDecode()` to convert a JSON string into a `Map<String, dynamic>`.
- Use `jsonEncode()` to convert a Dart object back into a JSON string.
- These functions make it easy to parse and generate JSON data in your Flutter apps.

Example:

```
Map<String, dynamic> data = jsonDecode('{"id": 1, "name": "Alberto"}');
```

JSON - Manual Model Class

- **Encapsulate parsing logic** within the model class for clarity and maintainability.
- **Requirements:**
  - Use a **private constructor** to control object creation.
  - Provide a `factory Person.fromJson(Map<String, dynamic> json)` for deserialization.
  - Implement a `Map<String, dynamic> toJson()` method for serialization.
- **Benefits:**
  - Ensures type safety.
  - Keeps parsing code organized and easy to update as your API evolves.
  - Promotes reusable and testable code.

JSON - Manual Model Class Example

```dart
class Person {
    final int id;
    final String value;

    Person._({required this.id, required this.value});

    factory Person.fromJson(Map<String, dynamic> json) =>
        Person._(id: json["id"], value: json["name"]);

    Map<String, dynamic> toJson() => {"id": id, "name": value};
}
```

- Use `Person.fromJson()` to create a `Person` from JSON.
- Use `toJson()` to convert a `Person` back to JSON.

JSON - Parsing Lists

- When decoding a JSON array, `jsonDecode()` returns a `List<dynamic>`.
- To convert each item to a model object, use `.map()` with your model's `fromJson` method:

```dart
final List<dynamic> jsonList = jsonDecode(jsonString);
final List<Person> people = jsonList
        .map((item) => Person.fromJson(item as Map<String, dynamic>))
        .toList();
```

- This approach ensures each element in the list is strongly typed as `Person`.
- Always cast each item to `Map<String, dynamic>` before passing to `fromJson`.
- Useful for parsing API responses that return lists of objects.

JSON – Complex Nested Objects

- **Nested JSON** means JSON objects containing other objects or lists.
- Example: An object with a `type` field and a `data` field (which is a list of `Person`).
- For each nested level, create a Dart model with `fromJson` / `toJson` methods.
- Manual parsing gets complex as nesting increases.
- Use tools like `json_serializable` to auto-generate code and avoid errors.
- This keeps your models simple and easy to update as APIs change.

JSON - Code Generation with `json_serializable`

- Add dependencies in `pubspec.yaml`:
  - `json_annotation` (for annotations)
  - `json_serializable` (for code generation)
  - `build_runner` (for running the generator)

```
dependencies:
  json_annotation: ^4.8.1

dev_dependencies:
  build_runner: ^2.4.6
  json_serializable: ^6.7.1
```

JSON - Automatic Model with `@JsonSerializable`

Example:

```
import 'package:json_annotation/json_annotation.dart';
part 'person.g.dart';

@JsonSerializable()
class Person {
    final int id;
    final String name;
    Person(this.id, this.name);

    factory Person.fromJson(Map<String, dynamic> json) =>
            _$PersonFromJson(json);
    Map<String, dynamic> toJson() => _$PersonToJson(this);
}
```

JSON - Generating Code with `build_runner`

- Generate code with:

```
flutter pub run build_runner build
```

or watch for changes with:

```
flutter pub run build_runner watch
```

- The generator creates `*.g.dart` files with `fromJson` and `toJson` methods automatically.
- This approach reduces boilerplate and keeps your models in sync with your API.

Automatic Model Class Example

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

    Person(this.id, this.name);

    factory Person.fromJson(Map<String, dynamic> json) =>
        _$PersonFromJson(json);
    Map<String, dynamic> toJson() => _$PersonToJson(this);
}
```

- No need to manually write parsing logic!

Example model (`post.dart`):

```
@JsonSerializable()
class Post {
    final int userId, id;
    final String title, body;
    const Post(this.userId, this.id, this.title, this.body);

    factory Post.fromJson(Map<String, dynamic> json) =>
            _$PostFromJson(json);
    Map<String, dynamic> toJson() => _$PostToJson(this);
}
```

- Run `build_runner` to generate `post.g.dart` with serialization logic.
- This keeps your code clean and reduces manual errors.

JSON - Generic Parser

Define a Base Parser

A **Base Parser** is an abstract class that defines a method for parsing JSON strings into Dart objects. This allows you to create specific parsers for different models by extending this base class.

```
abstract class JsonParser<T> {
    const JsonParser();
    Future<T> parseFromJson(String json);
    Future<T> parseToJson(T object) async {
        final Map<String, dynamic> data = (object as dynamic).toJson();
        return Future.value(data as T);
    }
}
```

Create a Parser for Your Model

A **Post Parser** is a concrete implementation of the `JsonParser` for the `Post` model. It uses the `fromJson` method of the `Post` class to convert a JSON string into a `Post` object.

```
class PostParser extends JsonParser<Post> {
    const PostParser();
    @override
    Future<Post> parseFromJson(String json) async {
        // Convert the JSON string to a Map and create a Post object
        final Map<String, dynamic> data = jsonDecode(json);
        return Post.fromJson(data);
    }
}
```

- Use `jsonDecode(json)` to convert the JSON string into a Dart map.
- Then use your model's `fromJson` method to create the object.

List Decoder Mixin (Simplified):

This mixin helps you easily convert a JSON string into a Dart `Map<String, dynamic>`, which is useful when working with JSON objects.

```
mixin ObjectDecoder<T> on JsonParser<T> {
    // Converts a JSON string into a Dart map
    Map<String, dynamic> decodeJsonObject(String json) {
        return jsonDecode(json) as Map<String, dynamic>;
    }
    String encodeJsonObject(Map<String, dynamic> object) {
        return jsonEncode(object) as String;
    }
}
```

- Use this mixin in your parser classes to quickly decode JSON objects.
- Makes your code cleaner and easier to maintain.

JSON - Example Model and Parser (`todo.dart`)

Suppose you have a `Todo` model and want to parse a list of todos from a JSON string.

Step 1: Create a Parser for a List of Todos

```
class TodoParser extends JsonParser<List<Todo>> with ListDecoder<List<Todo>> {
    @override
    Future<List<Todo>> parseFromJson(String json) async {
        // Decode the JSON string into a list of maps, then convert each map to a Todo object
        return decodeJsonList(json)
            .map((item) => Todo.fromJson(item as Map<String, dynamic>))
            .toList();
    }
}
```

- `decodeJsonList(json)` converts the JSON string into a Dart `List`.
- `.map((item) => Todo.fromJson(...))` turns each item into a `Todo` object.

Step 2: Fetch and Display Todos

```
late final Future<List<Todo>> todos;

@override
void initState() {
    super.initState();
    todos = RequestREST().executeGet<List<Todo>>(TodoParser(), "/todos");
}
```

- Here, `RequestREST().executeGet` fetches the list of todos from the API using the `TodoParser`.
- The result is stored in a `Future<List<Todo>>` for later use.

```
@override
Widget build(BuildContext context) {
    return FutureBuilder<List<Todo>>(
        future: todos,
        builder: (context, snapshot) {
            if (!snapshot.hasData) return CircularProgressIndicator();
            final todoList = snapshot.data!;
            return ListView(
                children: todoList.map((t) => ListTile(title: Text(t.title))).toList(),
            );
        },
    );
}
```

- Use `FutureBuilder` to show loading, error, or the list of todos.
- This pattern helps you fetch and display data from an API easily.

TIP: Using `@JsonKey`

The `@JsonKey` annotation lets you control how fields are mapped to JSON.

```
@JsonSerializable()
class User {
    final int id;
    @JsonKey(name: 'user_name') final String username;
    @JsonKey(defaultValue: false) final bool isActive;
    @JsonKey(ignore: true) final String? password; // Not in JSON

    User(this.id, this.username, this.isActive, this.password);

    factory User.fromJson(Map<String, dynamic> json) => _$UserFromJson(json);
    Map<String, dynamic> toJson() => _$UserToJson(this);
}
```

- `name` : Maps a JSON key to a different Dart field name.
- `defaultValue` : Sets a default if the key is missing in JSON.

END