part 2

Introduction to API’s

Server and Client in API Communication

Server: A server is a powerful system where data, such as images, resources, and services, is stored and managed. It is responsible for handling requests from clients and delivering the appropriate data or services in response. Examples of servers include the ones used by platforms like Google, Foodpanda, Facebook, and WhatsApp.

Client: A client is an end-user or a device that interacts with a server to access the data or services provided. The client's related information, such as preferences or account details, is stored on the server. When you use applications like YouTube, Facebook, or WhatsApp, your device acts as a client, sending requests to the server for data.

What is an API (Application Programming Interface)?

An API acts as a middleware that facilitates communication between a client and a server. It serves as a medium through which information can be requested from the server (using GET requests), submitted to the server (using POST requests), updated (using PUT or PATCH requests), or deleted (using DELETE requests). APIs enable seamless data transfer and interaction between the client and server.

How Does an API Work?

1. **Request:** When you search for a video on YouTube, your client device sends a request to the YouTube server.
2. **Validate:** The server receives the request and validates it to ensure that it is correctly formatted and authorized.
3. **Fetch:** The server then interacts with the database, fetching the relevant data (in this case, videos matching your search query).
4. **Return:** The server sends the requested data back to the client through the API, displaying the search results on your device.

In summary, the process involves the following steps:

Client Request → Server Validation → Database Fetching → API Response (Return)

This cycle allows clients to interact with servers, retrieving, submitting, updating, or deleting data as needed.

What is a REST API?

A REST API (Representational State Transfer Application Programming Interface) is an architectural style that defines a set of rules for building web services that allow

communication between a client and a server. RESTful APIs are designed to work with the HTTP protocol, which is commonly used on the web.

REST is based on a Client-Server (CS) approach, which means that the front end (client) and the back end (server) of an application are separated. This separation allows the client and server to operate independently, making REST one of the most popular approaches for building web services.

Key HTTP Methods Used in RESTful Services

1. **GET:**
   - **Purpose:** To retrieve data from the server.
   - **Example:** When you search for "Asif Taj tutorials" on YouTube, the GET request is sent to the server, which then returns the search results to you. The URL is unique and is used to fetch data from the server.
2. **POST:**
   - **Purpose:** To send data to the server to create or update a resource.
   - **Example:** In a login system, when you enter your email and password, a POST request is sent to the server. The server then validates your credentials against the database and returns a response (e.g., access to your account).
3. **PUT:**
   - **Purpose:** To update an existing resource on the server.
   - **Example:** If you have a list of products and you want to update a specific product's information, you would use a PUT request.
4. **DELETE:**
   - **Purpose:** To remove a resource from the server.
   - **Example:** If you want to delete a product from your list, a DELETE request is sent to the server to remove that item.

How REST APIs Work

When a client makes a request to the server via a REST API, the server processes the request, interacts with the database if necessary, and returns a response to the client. The response can include data (in the case of a GET request), confirmation of an action (like a POST or PUT request), or a notification that something was deleted (DELETE request).

In summary, REST APIs provide a structured way for clients and servers to communicate, using standard HTTP methods to perform various operations on resources stored on the server.

API (communication bw C & S , validate requests and return corresponding response)

part 3

Understanding JSON Structure

What is JSON (JavaScript Object Notation)?

- **Definition:** JSON stands for JavaScript Object Notation. It is a lightweight, text-based data-interchange format that is easy for humans to read and write and easy for machines to parse and generate.
- **Purpose:** JSON is commonly used to send data between computers, especially between a client and a server in web applications.
- **Language Independent:** While JSON is based on JavaScript syntax, it is language-independent, meaning it can be used with most programming languages like Python, Java, C#, and many others.

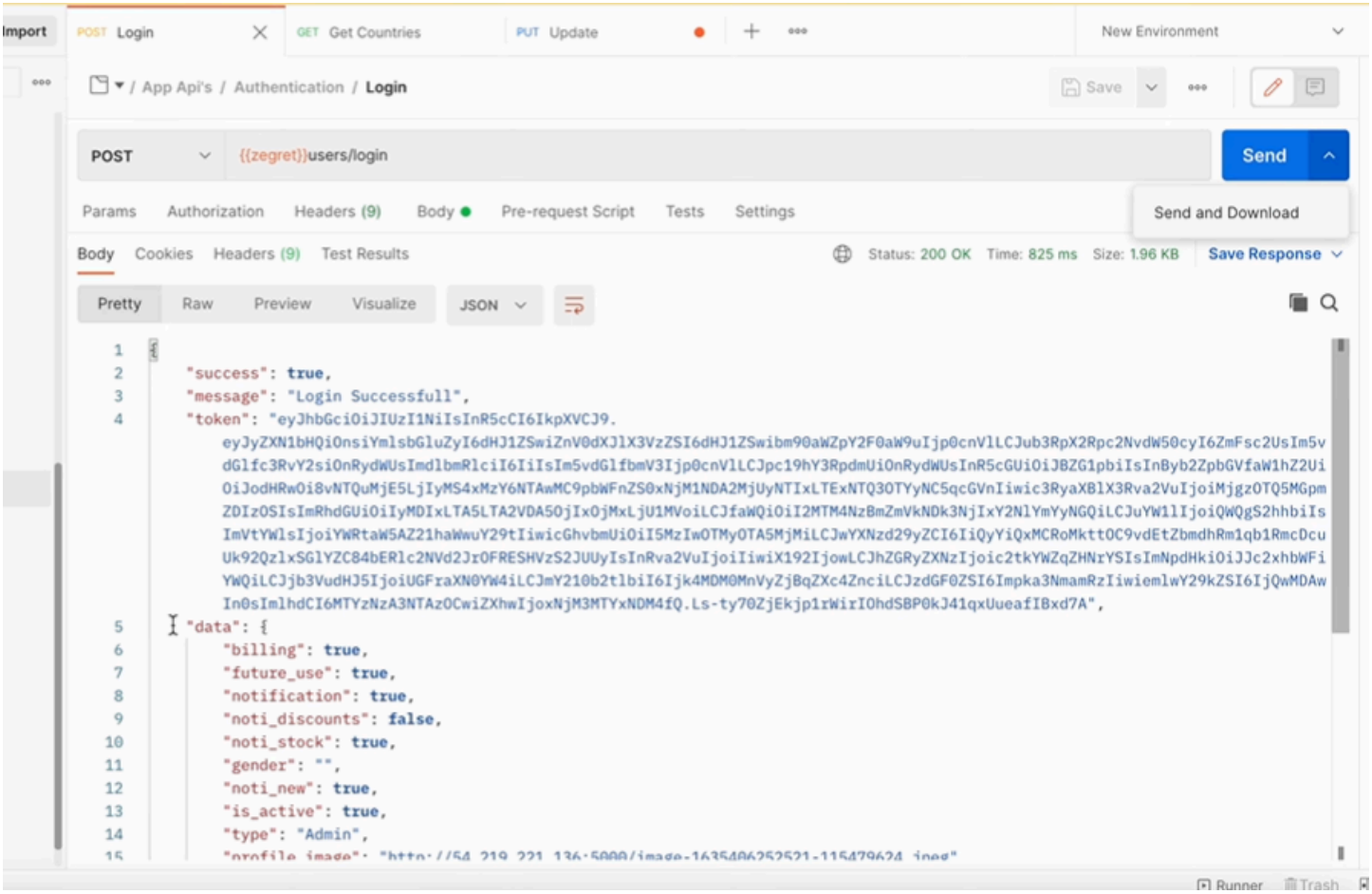
JSON Structure

JSON data is written as key-value pairs separated by commas. The basic structure of JSON is composed of:

1. **Key:** The key is always a string and represents the identifier for a value.
2. **Value:**The value can be one of several data types:
   - **String:** Enclosed in double quotes, e.g., "name": "John".
   - **Number:** An integer or floating-point number, e.g., "age": 30.
   - **Object:** A JSON object, which is a collection of key-value pairs enclosed in curly braces {}, e.g., "address": {"city": "New York", "zip": "10001"}.
   - **Array:** An ordered list of values enclosed in square brackets [], e.g., "colors": ["red", "green", "blue"].
   - **Boolean:** A value of true or false, e.g., "isStudent": true.
   - **Null:** Represents a null value, e.g., "middleName": null.

What is a JSON Object?

- **Definition:** A JSON object is a container for key-value pairs, enclosed in curly braces {}. The key is always a string, and the value can be any of the JSON data types.
-



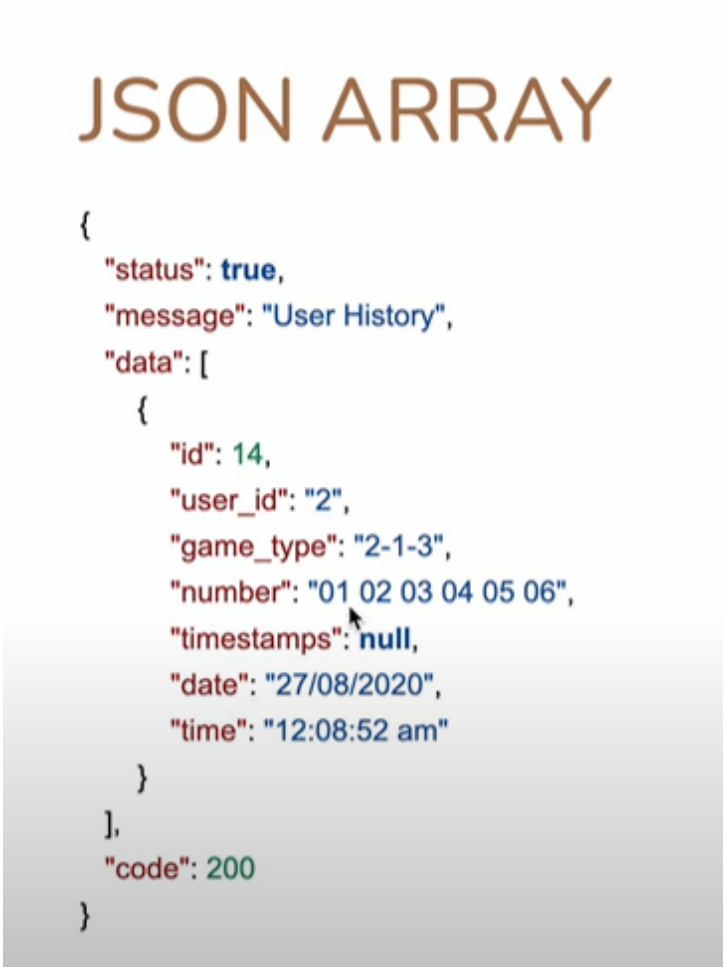
Nested JSON Object

Nested Json Object example:

```
{
  "status": true,
  "message": "Login Successfull",
  "data": {
        "name": "Salman",
         "email": "smkhanuet@gmail.com",
    "age": "26",
    },
  }
```

What is a JSON Array?

- **Definition:** A JSON array is an ordered list of values, enclosed in square brackets []. The values within an array can be of any data type, including objects and other arrays.



part 4

What is Postman

Postman

Postman is a popular software tool used for testing APIs. It allows developers to send requests to APIs, inspect responses, and ensure that the APIs are functioning as expected. Postman provides a user-friendly interface to interact with APIs without needing to write code, making it easier to understand how an API works and to test different endpoints.

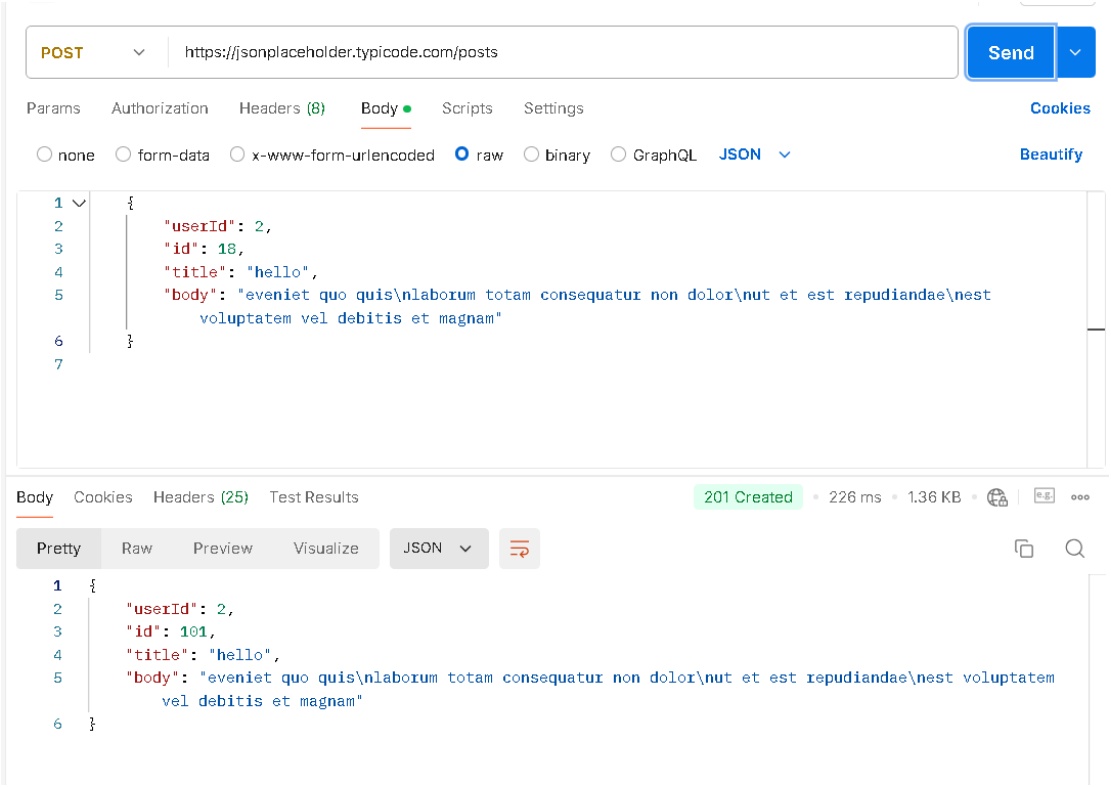
- **Testing APIs**: You can send various types of HTTP requests (GET, POST, PUT, DELETE) to your API, specify parameters, headers, and body content, and then view the response returned by the server.
- **Understanding JSON Structure**: Postman also helps you understand the structure of the data returned by the API, particularly when dealing with JSON. It displays the JSON response in a readable format, allowing you to inspect objects, arrays, and other data types.

JSONPlaceholder

JSONPlaceholder is a free online service that provides fake APIs for testing and prototyping. It's an excellent resource for developers who need to practice making API calls without

setting up a backend server.

- **Free APIs for Testing**: JSONPlaceholder offers several ready-to-use API endpoints that return JSON data. These endpoints mimic common API scenarios, such as fetching user data, posts, comments, and more.
- **Example Use Case**: If you're learning how to make API requests, you can use JSONPlaceholder to test your skills. For instance, you can make a GET request to https://jsonplaceholder.typicode.com/posts to retrieve a list of posts, or a POST request to create a new post.



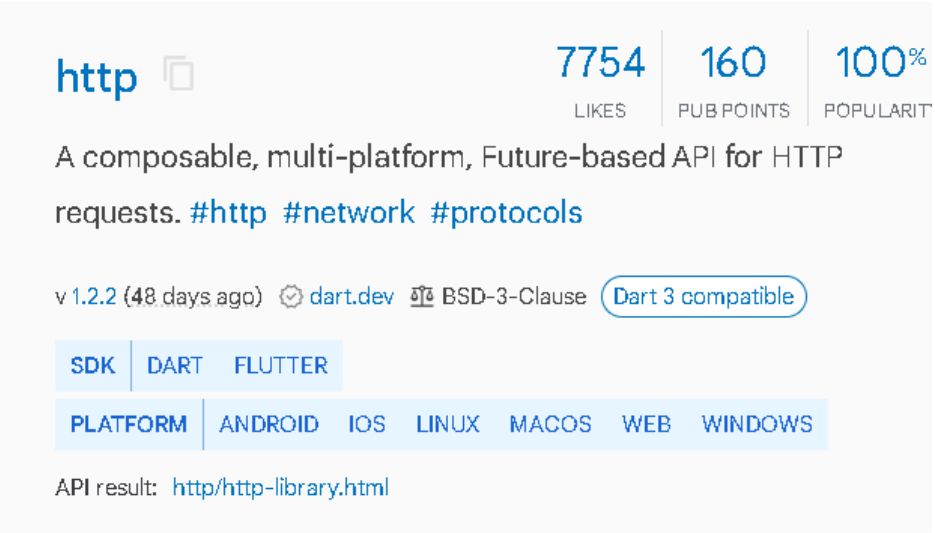
part 5

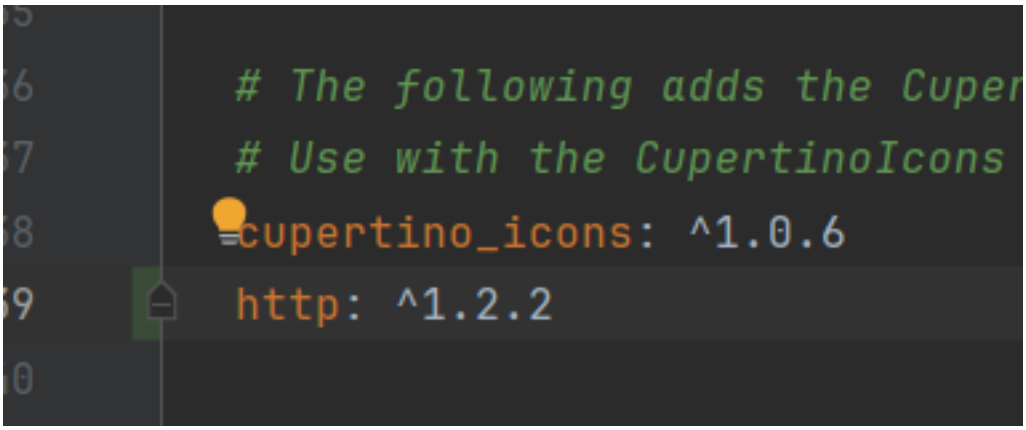
Creating flutter project with REST API

In Flutter, the http package is used to handle networking and API integration. It allows you to send HTTP requests (e.g., GET, POST) to a server and retrieve data, such as JSON, which can then be used in your app.

Steps to Use the http Package:

1. **Add Dependency**: Include the http package in your pubspec.yaml file.
2. **Import the Package:** In your Dart file, import the http package.
3. **Send a Request:** Use the http methods to send requests to the API.
4. **Handle the Response:** Parse the data returned from the server and use it in your Flutter app.

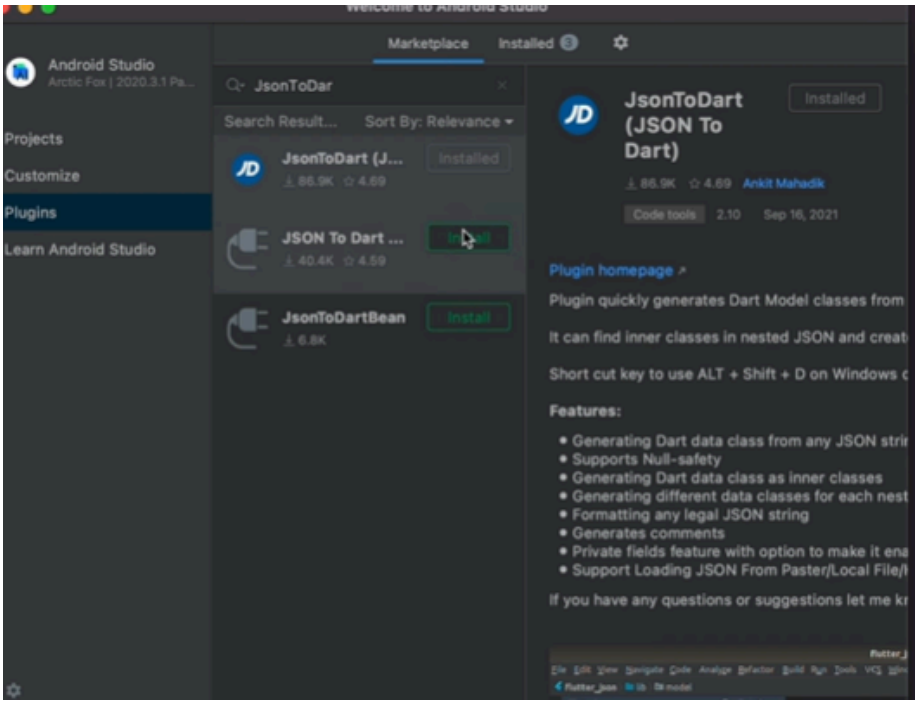




part 6

Convert Complex JSON to Dart Classes

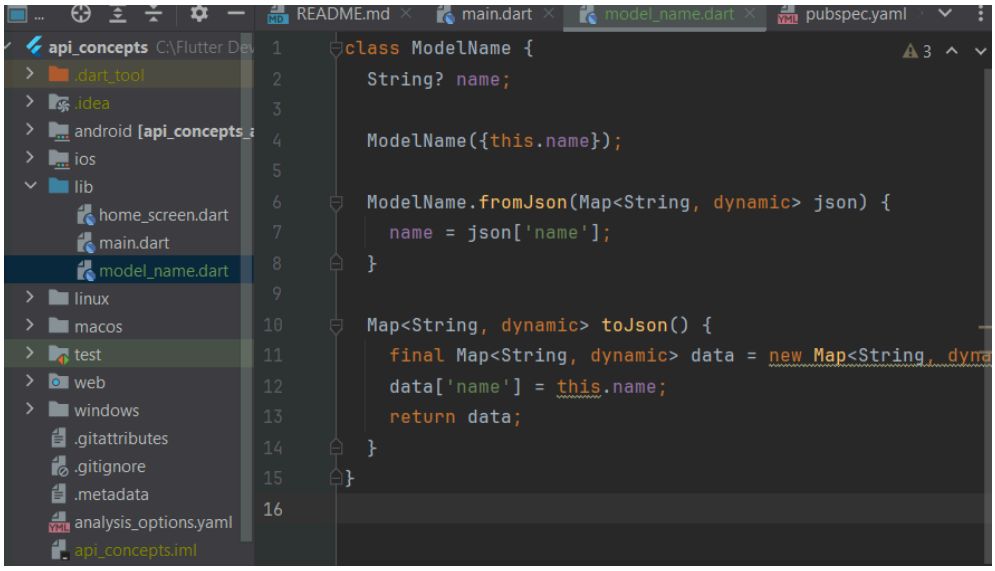
The jsondart plugin is a useful tool in Flutter that automatically generates model classes from JSON responses. Instead of manually writing custom code to parse JSON and create data models, jsondart allows you to simply provide the JSON structure, and it will generate the necessary Dart classes for you. This helps in streamlining the process of working with API responses, saving time, and reducing the likelihood of errors.



https://javiercbk.github.io/json_to_dart/

using above tool =>

then create a new dart file in flutter project and paste the class there.



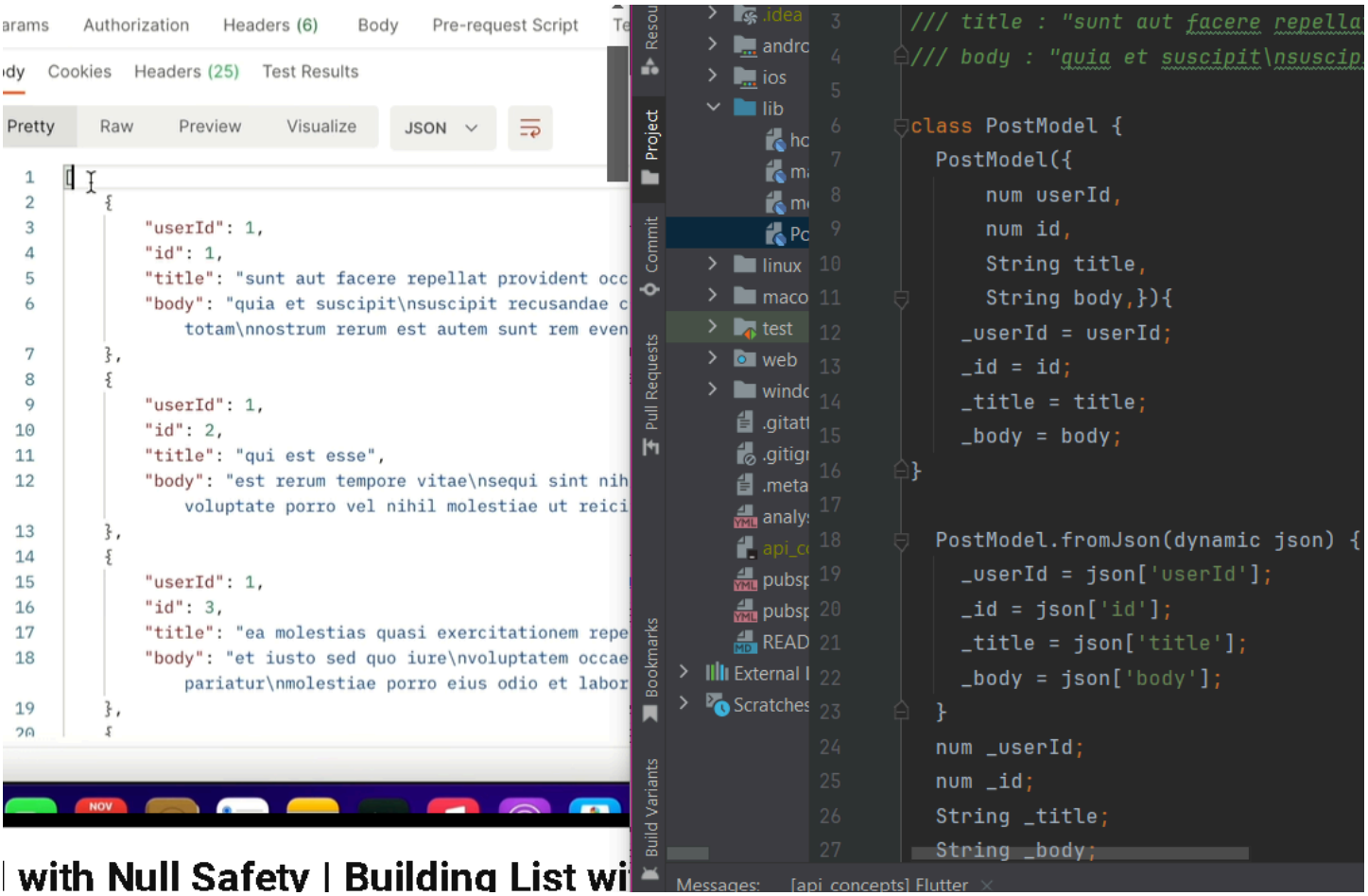
part 7

Flutter Get API call with Null Safety | Building List with JSON Data using Future Builder

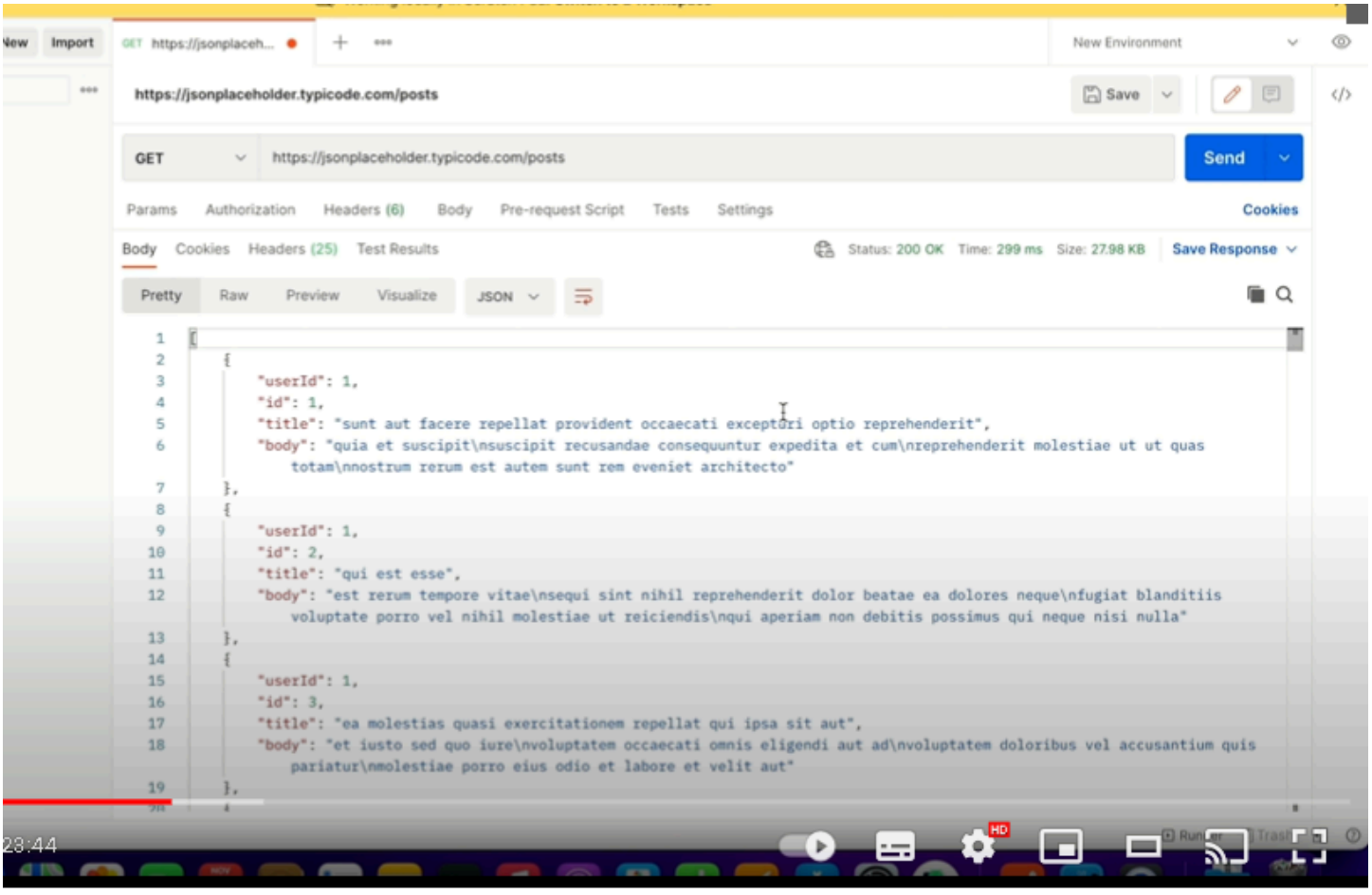
Create the Model: Use a tool like jsondart to automatically generate a Dart model from the JSON response structure.

Parse the JSON Response: When you receive the JSON response from the API, you parse it and map the data to your model.

Map the Data into a List: If the JSON response contains a list of items, you can map each item to an instance of your model and store it in a Dart list.



with Null Safety | Building List wi



Steps to Fetch and Parse API Data:

- **Create a List to Store the Data:**

```
List<PostsModel> postList = [];
```

Here, PostsModel is the model class that will store the data fetched from the API. The postList is initialized as an empty list.

- **Create an Asynchronous Function to Fetch Data:**

```
Future<List<PostsModel>> getPostApi() async {
```

This function will return a Future that eventually contains a List of PostsModel. The async keyword indicates that the function will perform an asynchronous operation.

- **Send the HTTP GET Request:**

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

This line sends an HTTP GET request to the provided URL and waits for the response.

- **Decode the JSON Data:**

```
var data = jsonDecode(response.body.toString());
```

The response body, which is in JSON format, is decoded into a Dart object. The jsonDecode function is used to parse the JSON string into a map or list, depending on the structure of the JSON.

- **Check for a Successful Response:**

```
if (response.statusCode == 200) {
```

The status code 200 indicates a successful request. If the request is successful, the following code will execute.

- **Iterate Over the JSON Data:**

```
for (Map i in data) {
  postList.add(PostsModel.fromJson(i));
}
```

Here, the code loops over each item in the data (which is expected to be a list of maps). Each item is added to postList after being converted into a PostsModel object using the fromJson method.

- **Return the List:**

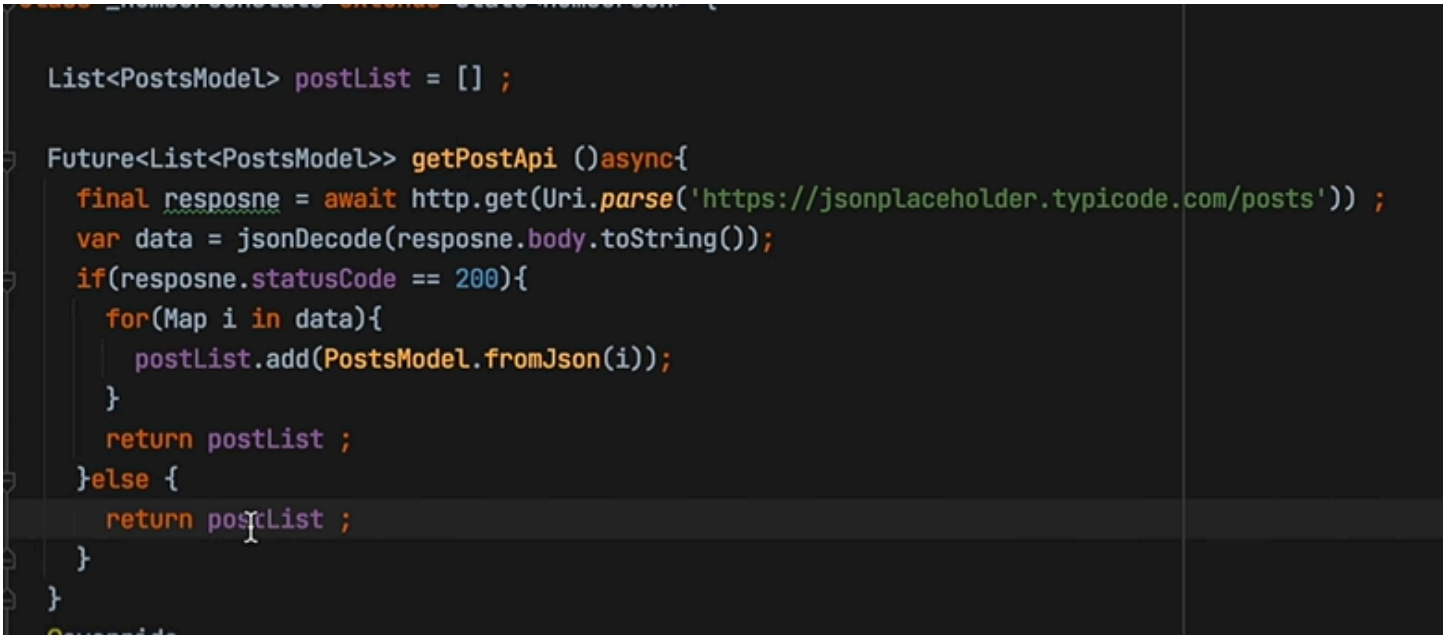
return postList;

The list postList, now populated with PostsModel objects, is returned from the function.

- **Handle Failure:**

```
else {
  return postList;
}
```

If the response status is not 200, the function will still return an empty postList.



```
List<PostsModel> postList = [] ;

Future<List<PostsModel>> getPostApi ()async{
  final resposne = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts')) ;
  var data = jsonDecode(resposne.body.toString());
  if(resposne.statusCode == 200){
    for(Map i in data){
      postList.add(PostsModel.fromJson(i));
    }
    return postList ;
  }else {
    return postList ;
  }
}
```

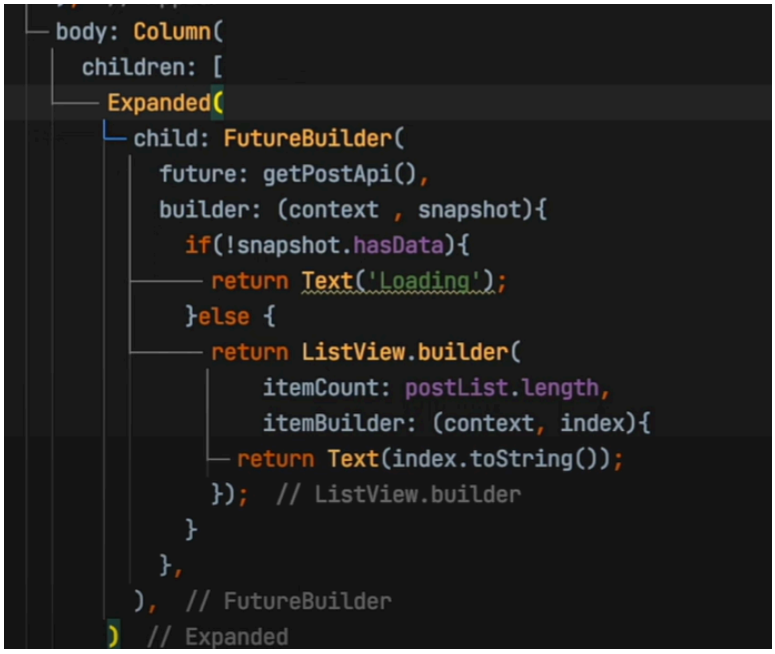
Steps to display data

To display data from an API, we use a FutureBuilder, which handles asynchronous data fetching and UI updates. The FutureBuilder takes a future that represents the API call (getPostApi in this case) and a builder function that provides the context and snapshot of the API response.

The FutureBuilder waits for the API response, and once it arrives, it checks if the snapshot contains data using snapshot.hasData. If the data is available, it displays the desired list using ListView.builder. If no data is available (i.e., the snapshot has no data), it shows a loading message (Text("Loading")).

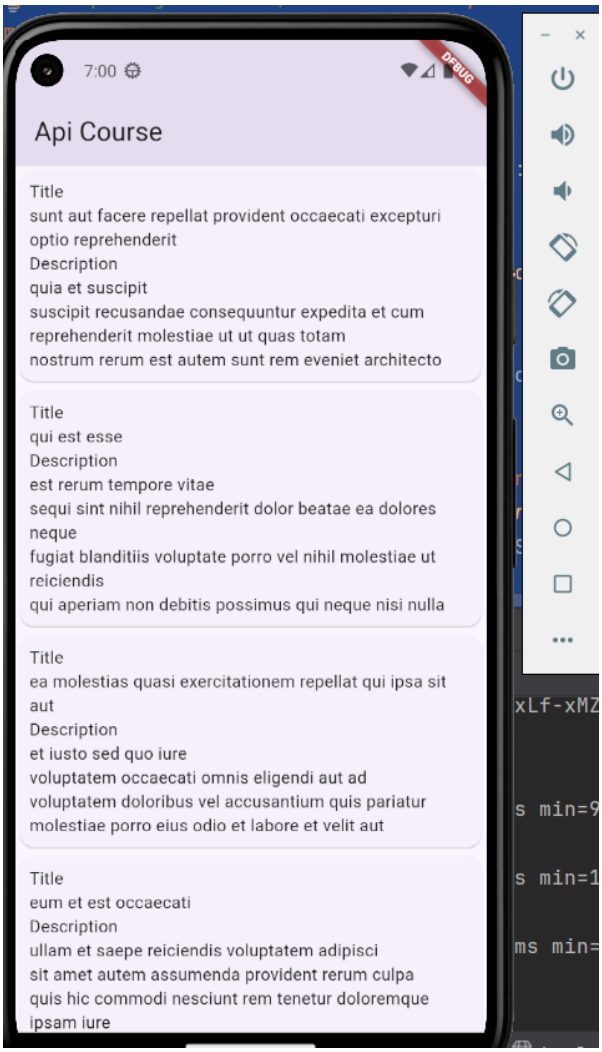
In the ListView.builder, we provide an itemCount, which is set to the length of the postList, and an itemBuilder function that co-nstructs each list item. Each item is displayed using a

Text widget that shows the index in this example, but it can be customized with a Card widget or any other layout as needed.




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

class HomeScreen extends StatefulWidget {
 const HomeScreen({Key? key}) : super(key: key);

 @override
 _HomeScreenState createState() => _HomeScreenState();
}

class _HomeScreenState extends State<HomeScreen> {
 List<PostModel> postList = [];

 Future<List<PostModel>> getPostApi() async {
  final response = await http.get(Uri.parse('https://jsonplaceholder.typicode.com/posts'));
  var data = jsonDecode(response.body.toString());
  if (response.statusCode == 200) {
   for (Map i in data) {
    postList.add(PostModel.fromJson(i as Map<String, dynamic>));
   }
   return postList;
  } else {
   return postList;
```

```
  }
 }

 @override
 Widget build(BuildContext context) {
  return Scaffold(
   appBar: AppBar(
    title: Text('Api Course'),
   ),
   body: Column(
    children: [
     Expanded(
      child: FutureBuilder(
       future: getPostApi(),
       builder: (context, snapshot) {
        if (!snapshot.hasData) {
         return Center(child: CircularProgressIndicator());
        } else {
         return ListView.builder(
          itemCount: postList.length,
          itemBuilder: (context, index) {
           return Card(
            child: Padding(
             padding: const EdgeInsets.all(8.0),
             child: Column(
              mainAxisAlignment: MainAxisAlignment.start,
              crossAxisAlignment: CrossAxisAlignment.start,
              children: [
               Text('Title\n' + postList[index].title.toString()),
               Text('Description\n' + postList[index].body.toString()),
              ],
             ),
            ),
           );
          },
         );
        }
       },
      ),
     ),
    ],
   ),
  );
 }
}
```



part 8

Building List with JSON Data with Custom Model

- ***Model Creation:*** *The Photos class models the structure of the JSON data.*
- ***Fetching Data:*** *The getPhotos function fetches data, converts it into Photos objects, and stores them in a list.*
- ***Handling Asynchronous Data:*** *The Future<List<Photos>> return type ensures that we handle the asynchronous nature of API calls, waiting for the data to be available before proceeding.*

This approach effectively handles JSON data in Flutter, allowing you to convert it into Dart objects for easier manipulation.

Creating a Custom Model in Flutter

Let’s say we have the following JSON data:
{

"albumId": 1,

"id": 1,

"title": "accusamus beatae ad facilis cum similique qui sunt",

"url": "https://via.placeholder.com/600/92c952",

"thumbnailUrl": "https://via.placeholder.com/150/92c952"

}

To work with this JSON, we first create a class to model the data structure. Here's how we can do it:

1. **Define the Class:**

First, we declare a class Photos to represent the JSON object.
class Photos {

String title, url;

Photos({required this.title, required this.url});

}
String title, url;: These are the fields we want to extract from the JSON.

- **Photos({required this.title, required this.url});**: This is the constructor that initializes the title and url fields when a Photos object is created.

Fetching Data from an API:

Now, we need a function that fetches the data from the API and converts the JSON response into a list of Photos objects.

```
List<Photos> photosList = []; // List of Photos objects

Future<List<Photos>> getPhotos() async {
 final response = await http.get(
   Uri.parse('https://jsonplaceholder.typicode.com/photos'));

 var data = jsonDecode(response.body.toString()); // Decoding the response

 if (response.statusCode == 200) { // If the request is successful (200 OK)
  for (Map i in data) {
   Photos photos = Photos(
    title: i['title'],
    url: i['url'],
```

```
      );
      photosList.add(photos); // Add each Photos object to the list
    }
    return photosList;
  } else {
    return photosList; // In case of an error, return an empty list
  }
}
```

List<Photos> photosList = [];: This is an empty list where the photos will be stored after being fetched.

Future<List<Photos>> getPhotos() async {...}: This is an asynchronous function that fetches data from the given API, decodes the JSON, and converts it into a list of Photos objects.

Inside the loop:

Photos photos = Photos(title: i['title'], url: i['url']); photosList.add(photos);

Here, we create an instance of Photos using the constructor, and we pass the title and url from the JSON. The Photos object is then added to the photosList.

Returning the List:
Once the data is processed, we return the list of photos. If the status code is 200 (indicating a successful request), the list will be populated with photos. Otherwise, we return the empty list.

Rendering Data from an API using FutureBuilder

Displaying the API Data

The API data is displayed using a FutureBuilder widget, which is a built-in Flutter widget that builds itself based on the latest snapshot of interaction with a Future.

syntax: **FutureBuilder<T>(**

{required Future<T> future,

required Widget Function(BuildContext context, AsyncSnapshot<T> snapshot) builder})

In this case, T is List<Photos>, which is the type of data returned by the getPhotos function.

```
child: FutureBuilder<List<Photos>>(
  future: getPhotos(),
  builder: (context, AsyncSnapshot<List<Photos>> snapshot) {
```

FutureBuilder Construction

Expanded(

```
 child: FutureBuilder<List<Photos>>(
  future: getPhotos(),
  builder: (context, AsyncSnapshot<List<Photos>> snapshot) {
   // Code to handle the snapshot
  },
 ),
)
```

Here, getPhotos() is the future that returns a list of Photos objects. The builder function is called whenever the snapshot changes.

Handling the Snapshot

The builder function takes two arguments: context and snapshot. context is the build context, and snapshot is an AsyncSnapshot object that contains the latest data from the future.

AsyncSnapshot Syntax

```
AsyncSnapshot<T>({
 ConnectionState connectionState,
 T data,
 Object error,
 StackTrace stackTrace
})
```

In this case, T is List<Photos>, which is the type of data returned by the getPhotos function.

ListView.builder Construction

```
return ListView.builder(
 itemCount: photoslist.length,
 itemBuilder: (context, index) {
  return ListTile(
```

```
    leading: CircleAvatar(
     backgroundImage: NetworkImage(
        snapshot.data![index].url.toString()),
    ),
    title: Text(snapshot.data![index].title.toString()),
    subtitle: Text("Notes id: " + snapshot.data![index].id.toString()),
   );
 },
);
```

Here, we create a ListView.builder with the following properties:

- itemCount: The number of items in the list, which is the length of the photoslist list.
- itemBuilder: A function that builds each list item.

ListTile Syntax

```
return ListTile(
 leading: CircleAvatar(
  backgroundImage: NetworkImage(
    snapshot.data![index].url.toString()),
 ),
 title: Text(snapshot.data![index].title.toString()),
 subtitle: Text("Notes id: " + snapshot.data![index].id.toString()),
);
```

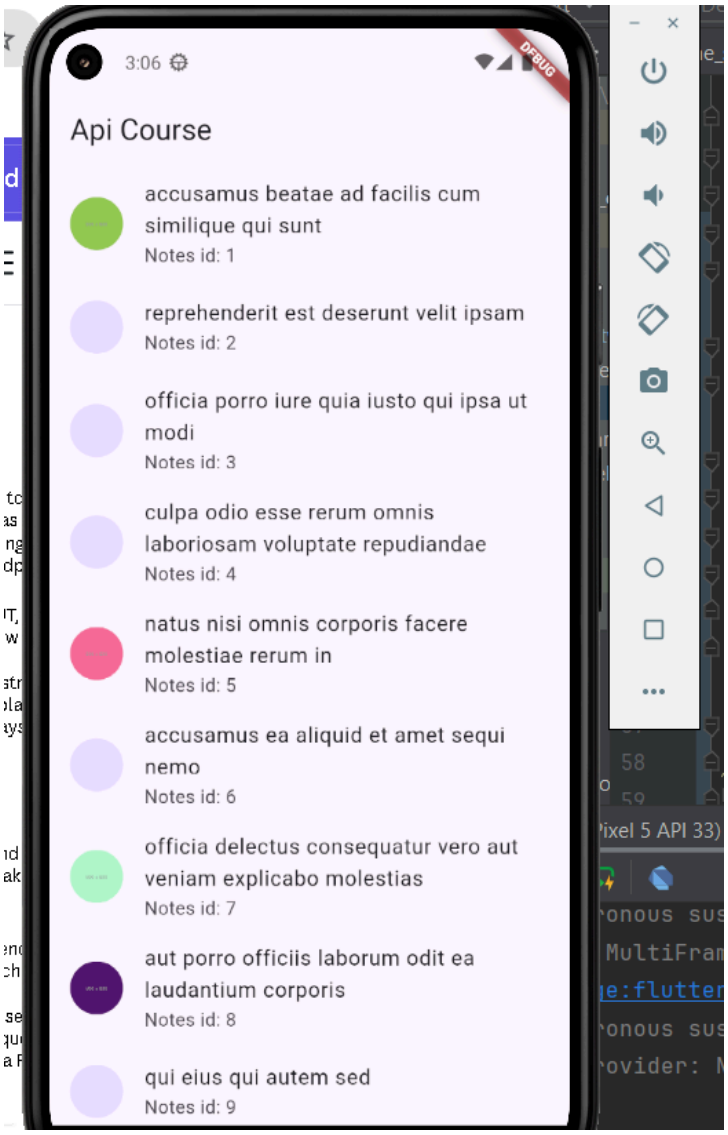
Here, we create a ListTile with the following properties:

- leading: A CircleAvatar widget that displays the photo URL.
- title: A Text widget that displays the photo title.
- subtitle: A Text widget that displays the photo ID.

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

class ExampleTwo extends StatefulWidget {
 const ExampleTwo({super.key});

 @override
 State<ExampleTwo> createState() => _ExampleTwoState();
}

class _ExampleTwoState extends State<ExampleTwo> {
 List<Photos> photoslist = []; //list is provided photos constructor
 Future<List<Photos>> getPhotos() async {
  //till the time we havent returned list in our if else statements getPhotos()func was showing error bec it was expecting a list to be returned
  final response = await http
    .get(Uri.parse('https://jsonplaceholder.typicode.com/photos'));
  var data = jsonDecode(response.body.toString());
  if (response.statusCode == 200) {
   for (Map i in data) {
    Photos photos = Photos(
      title: i['title'],
      id: i['id'],
      url: i[
        'url']); //photos is the name of the object you're creating. Photos(title: i['title'], url: i['url']) is the constructor of the Photos class that is being called with two arguments: title and url.
    photoslist.add(photos);
   }
   return photoslist;
   //200 means request is valid
  } else {
   return photoslist;
  }
 } //creating future function tht waits for future value and return type is list with its model
```

```
@override
Widget build(BuildContext context) {
 return Scaffold(
  appBar: AppBar(
   title: Text('Api Course'),
  ),
  body: Column(
   children: [
    Expanded(
     child: FutureBuilder<List<Photos>>(
       future: getPhotos(),
       builder: (context, AsyncSnapshot<List<Photos>> snapshot) {
        return ListView.builder(
          itemCount: photoslist.length,
          itemBuilder: (context, index) {
           return ListTile(
            leading: CircleAvatar(
             backgroundImage: NetworkImage(
               snapshot.data![index].url.toString()),
            ),
            title: Text(snapshot.data![index].title.toString()),
            subtitle: Text("Notes id: " +
              snapshot.data![index].id.toString()),
           );
          });
       }),
     )
    ],
   ),
  );
 }
}

class Photos {
 String title, url;
 int id;
 Photos({required this.title, required this.url, required this.id});
```

```
}
```



part 9

Building List with Complex JSON using FutureBuilder

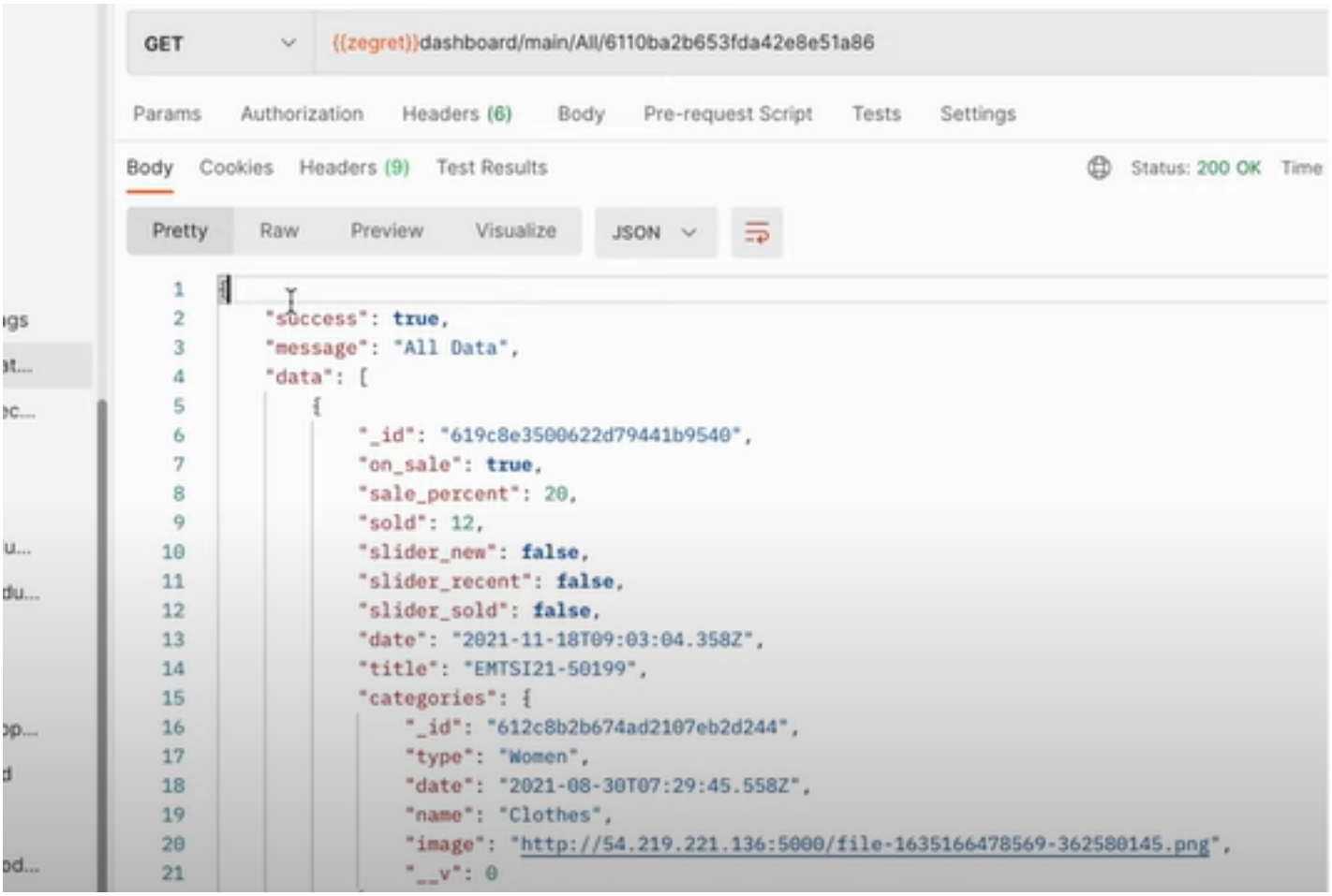
```
import 'dart:convert';
import 'package:flutter/material.dart';
import 'package:http/http.dart' as http;
import 'UserModel.dart'; // Make sure to import the UserModel

class ExampleThree extends StatefulWidget {
```

```dart
  const ExampleThree({super.key});

  @override
  State<ExampleThree> createState() => _ExampleThreeState();
}

class _ExampleThreeState extends State<ExampleThree> {
  List<UserModel> userList = [];

  // Fetching users from the API
  Future<List<UserModel>> getUserApi() async {
    final response =
    await http.get(Uri.parse('https://jsonplaceholder.typicode.com/users'));
    var data = jsonDecode(response.body.toString());

    // Checking if the response is OK
    if (response.statusCode == 200) {
      for (Map i in data) {
        userList.add(UserModel.fromJson(i));  // Parsing and adding user data
      }
      return userList;
    } else {
      return [];
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        title: const Text('API Course'),
      ),
      body: Column(
        children: [
          Expanded(
            child: FutureBuilder<List<UserModel>>(
              future: getUserApi(),
              builder: (context, AsyncSnapshot<List<UserModel>> snapshot) {
```

```
          // Show a loading indicator while fetching data
          // Display error message if something goes wrong
           if (snapshot.hasError) {
            return Center(child: Text('Error: ${snapshot.error}'));
          }
          // Display the list if data is available
          else if (snapshot.hasData && snapshot.data!.isNotEmpty) {
           return ListView.builder(
             itemCount: snapshot.data!.length,
             itemBuilder: (context, index) {
               var user = snapshot.data![index];
               return Card(
                 child: ListTile(
                   leading: CircleAvatar(
                     child: Text(user.id.toString()),
                   ),
                   title: Text(user.name ?? 'No Name'),
                   subtitle: Text(user.email ?? 'No Email'),
                   trailing: Text(user.address?.city ?? 'No City'),
                 ),
               );
             },
           );
          }
          // Display message if no data is available
          else {
           return const Center(child: Text('No Data Available'));
          }
        },
      ),
    ),
  ],
 ),
);
 }
}
```

part 10 (also covers part 11)

Building List with very Complex JSON



This JSON response is an object containing multiple nested structures. It starts as an object with several properties, one of which is an array of objects. Each object within the array has various properties, including another nested array ("categories").
An object containing arrays (like "data").Inside the array, there are objects (individual records/items).

🌐 Webhook.site - Test, transform and automate Web requests and emails

a platform where you can create fake apis

https://jsonviewer.stack.hu

To understand the structure of your API (such as data flow, hierarchy, and behavior of nested objects and arrays), you can approach it

Viewer | Text

Paste | Copy | Format | Remove white space | Clear | Load JSON data


```
'created_At" : "2023-01-05T07: 56: 19.219Z" ,
'name" :"Shirts" ,
'description" : "This is enderobe brand" ,
'shopemail" : "adenreobe@gmail.com" ,
'shopaddress" : "Islamabad F8" ,
'shopcity" : "Islamabad" ,
'userid" : "23asdfasfdsfd" ,
'image" : "https://images.pexels.com/photos/5531004/pexels-photo-5531004.jpeg?auto=compress&cs=tinysrgb&w=1200"
          },
'price" : "2500" ,
'sale_title" : "Winter Sale" ,
'sale_price" : "2300" ,
'description" : "22% off all the sales " ,
'color" : "#FF6347,#CD5C5C" ,
'size" : "S,M,L,XL,XXL" ,
'in_wishlist" : true,
'images" : [
                {
'id" : "2342asdfasdf34",
'url" : "https://images.pexels.com/photos/19090/pexels-photo.jpg?auto=compress&cs=tinysrgb&w=1260&h=750&dpr=2"
                },
```


Viewer | Text

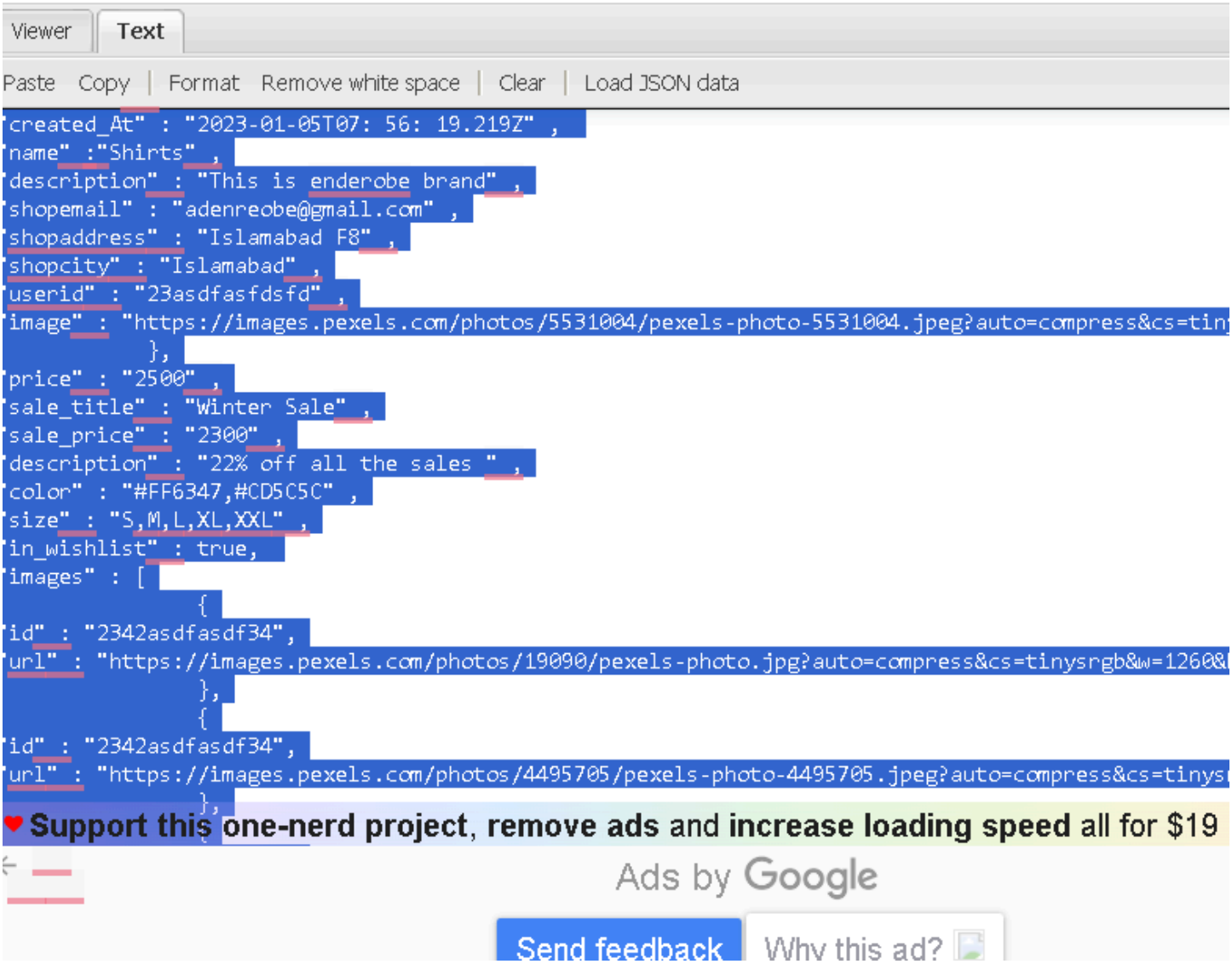
JSON
- success : true
- message : "All data"
- data

Name	Value
data	...
message	"All data"
success	true

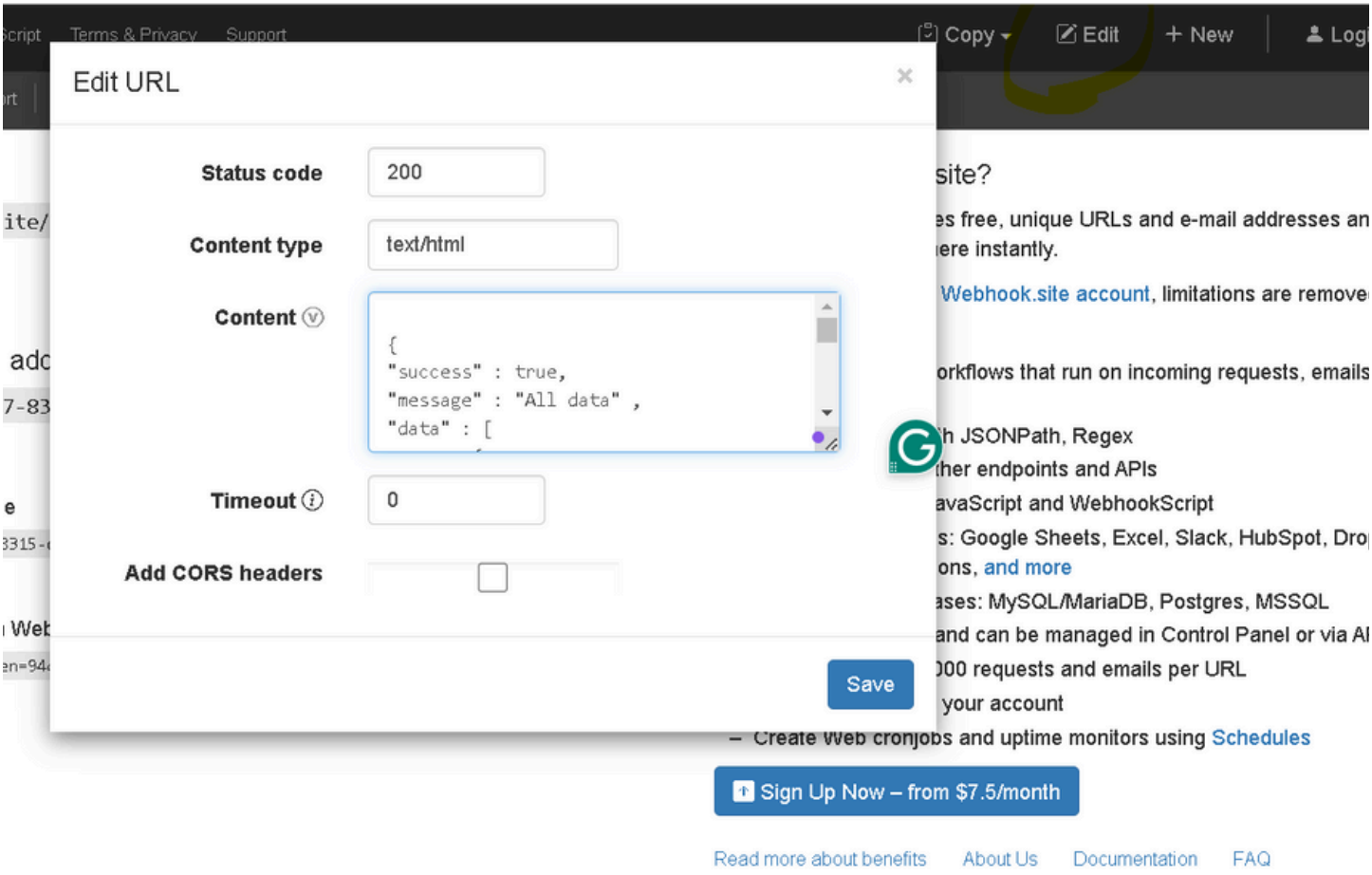
this basically tells us that we have object and in that object we have array of data then in data array we have 9 nested objects

then in individual objects we have parameters

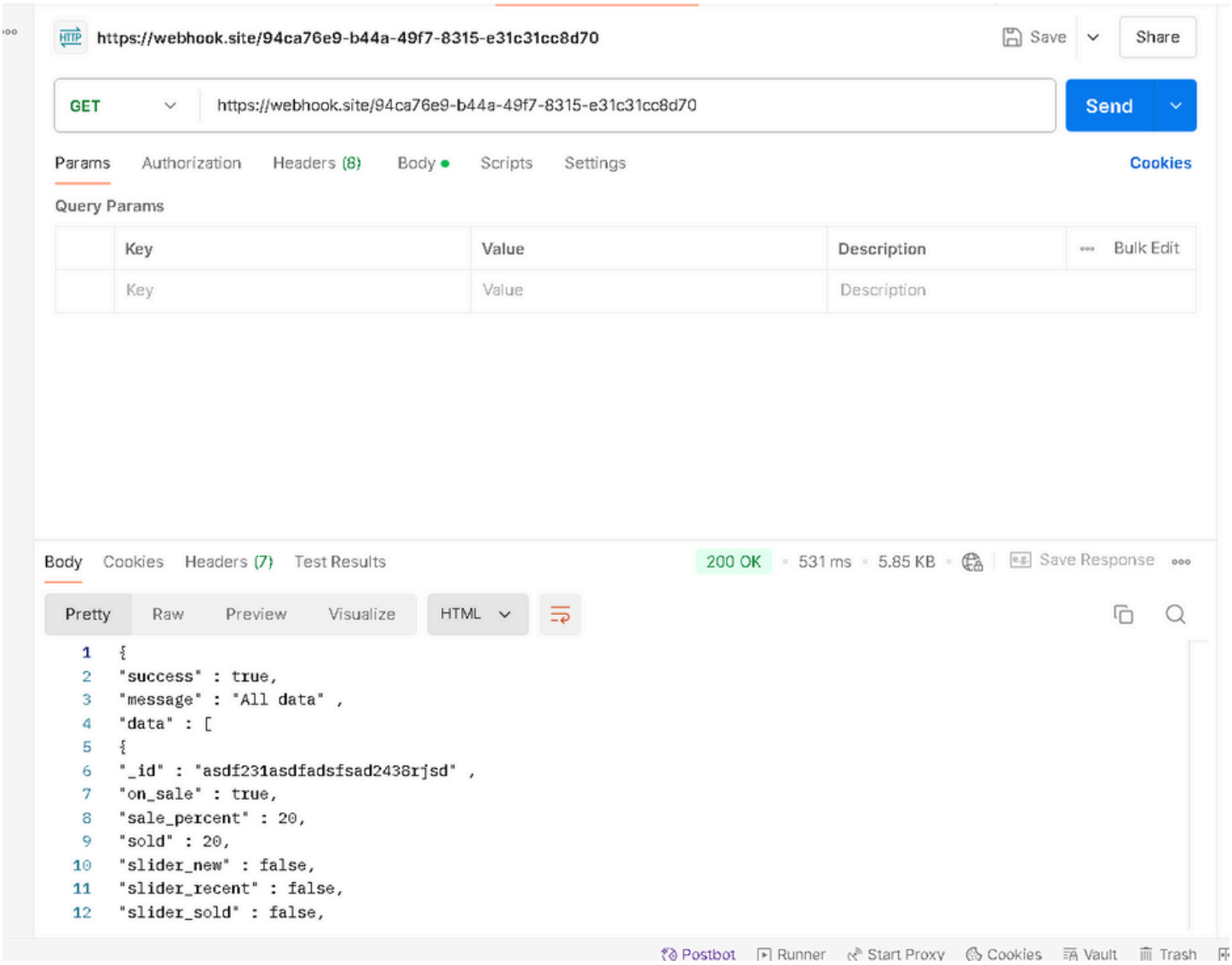
then copy the entire response



and paste it in response body in webhook to get the fake api



then copy the link the and paste it in postman

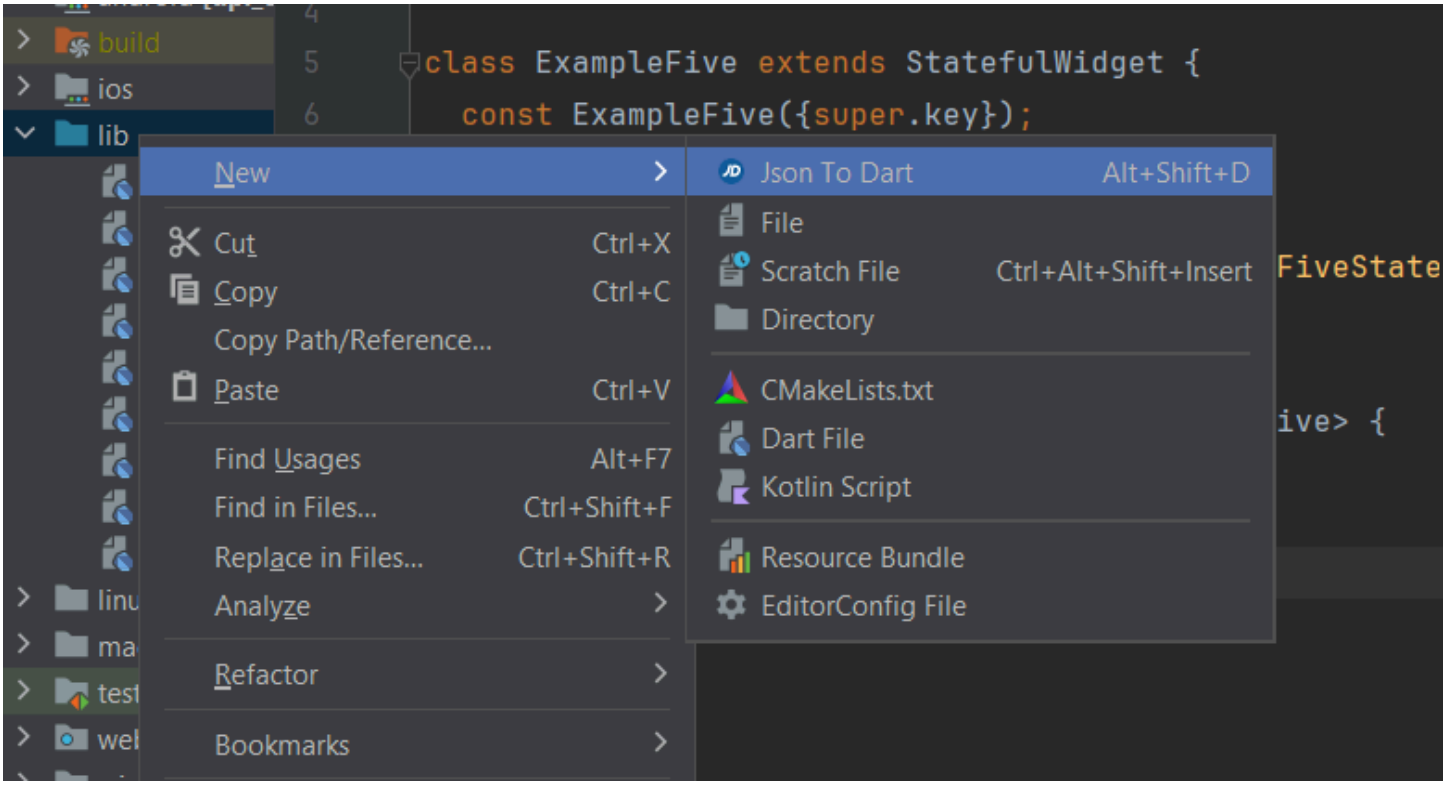


for model, copy the below response

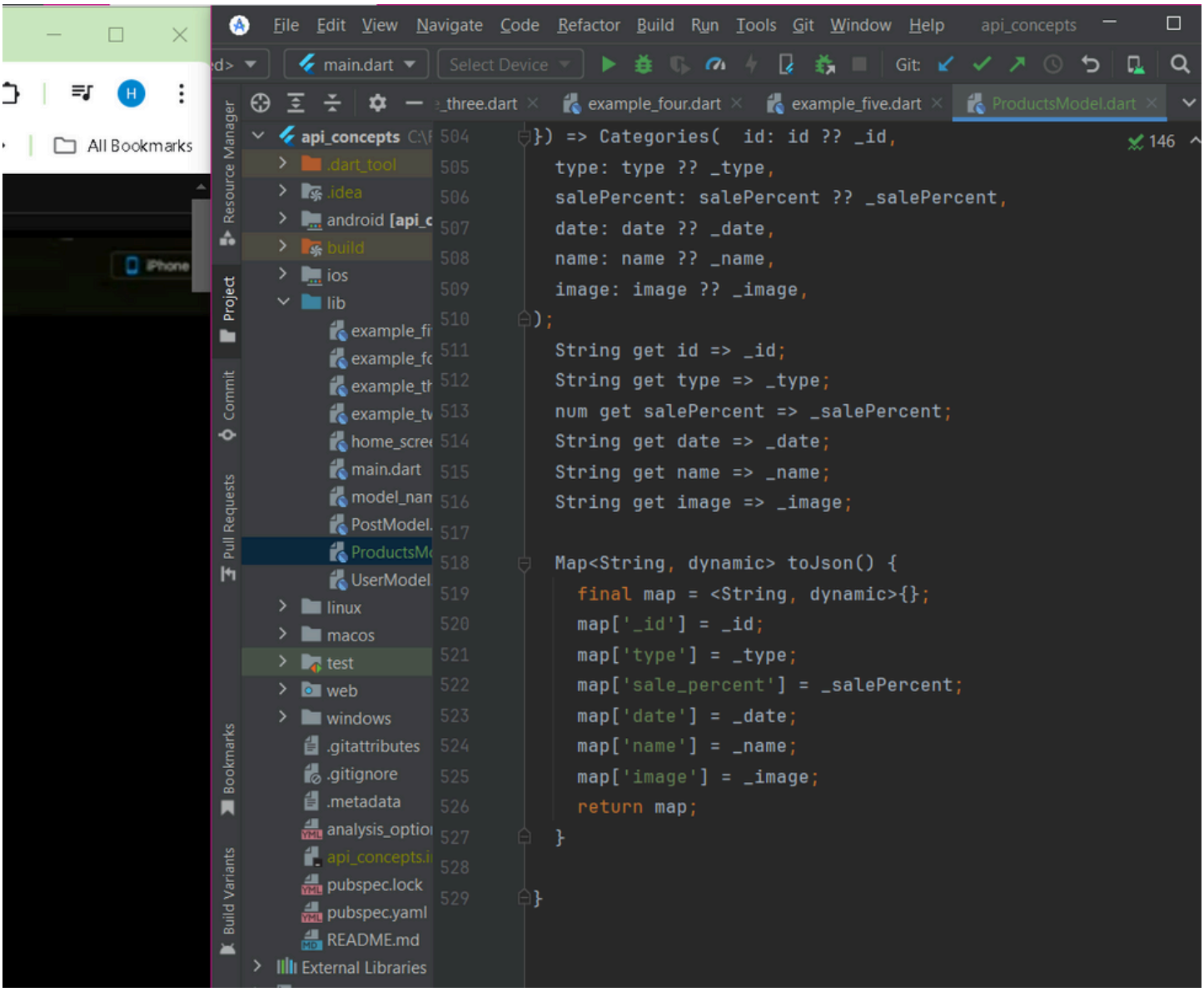
Body Cookies Headers (7) Test Results 200 OK

Pretty Raw Preview Visualize JSON

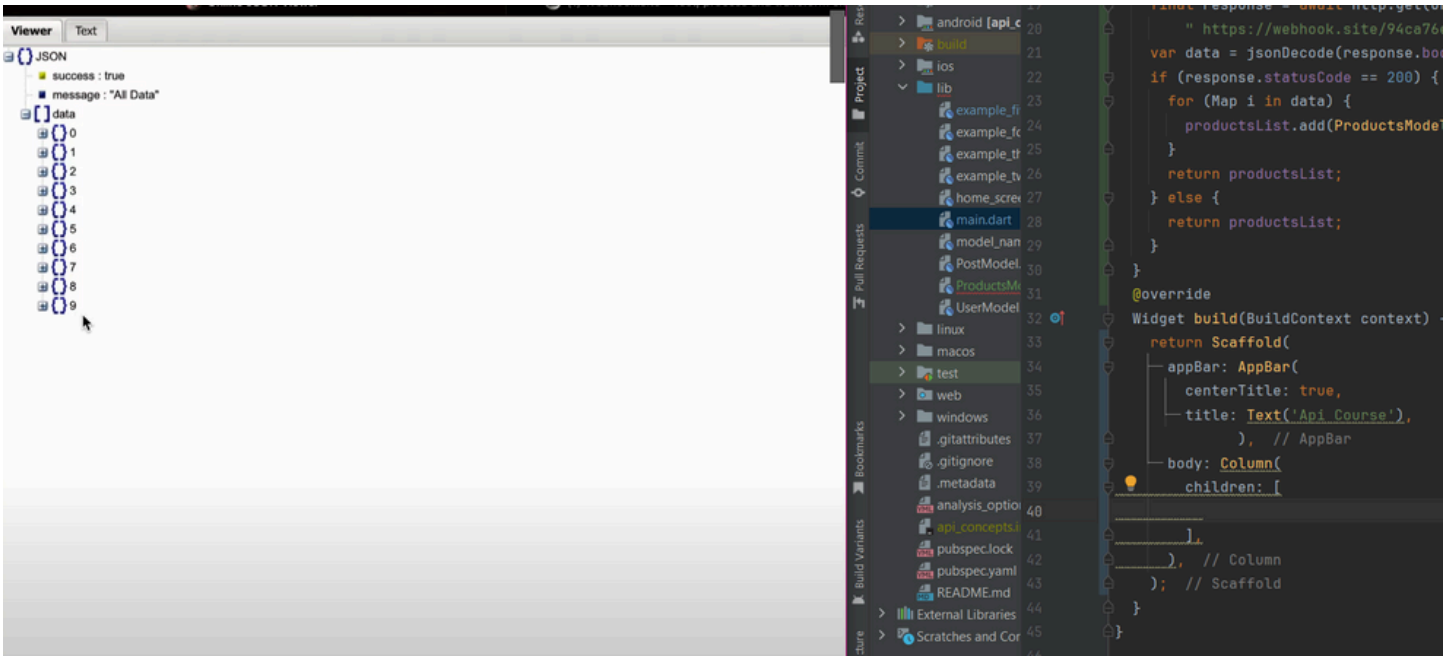

```
{
    "success": true,
    "message": "All data",
    "data": [
        {
            "_id": "asdf231asdfadsfsad2438rjsd",
            "on_sale": true,
            "sale_percent": 20,
            "sold": 20,
            "slider_new": false,
            "slider_recent": false,
            "slider_sold": false,
```

then complex model is ready



Based on the data we know that we've arrays, so we will use ListView.Builder

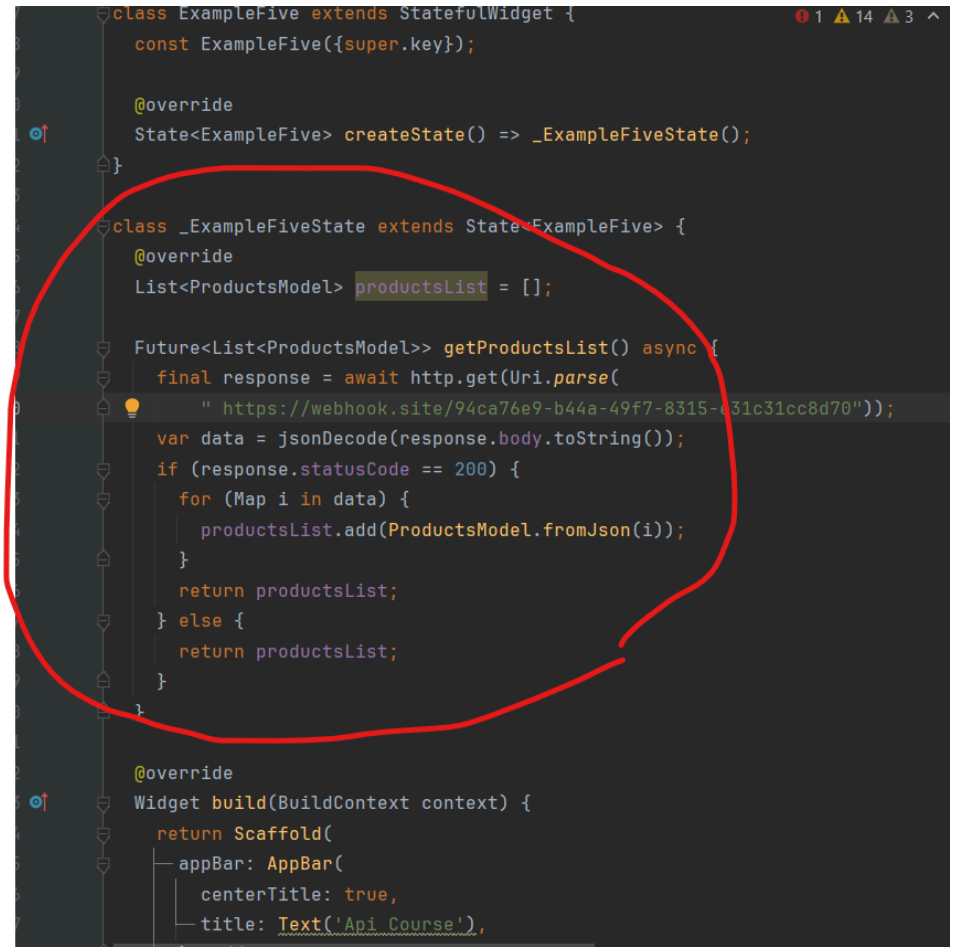


it is evident that the data is being build 9 times but in app we use the size of the list

Summing it all

first step =>

making a GET request to a REST API using the http package in Flutter.



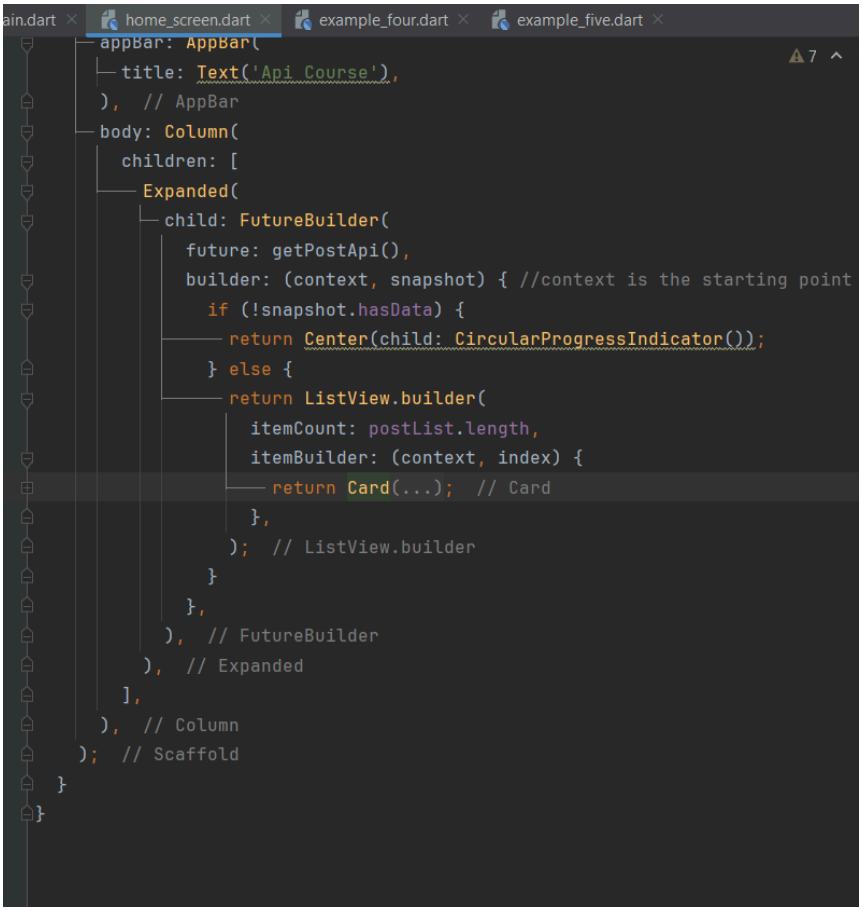
```dart
class ExampleFive extends StatefulWidget {
  const ExampleFive({super.key});

  @override
  State<ExampleFive> createState() => _ExampleFiveState();
}

class _ExampleFiveState extends State<ExampleFive> {
  @override
  List<ProductsModel> productsList = [];

  Future<List<ProductsModel>> getProductsList() async {
    final response = await http.get(Uri.parse(
        " https://webhook.site/94ca76e9-b44a-49f7-8315-e31c31cc8d70"));
    var data = jsonDecode(response.body.toString());
    if (response.statusCode == 200) {
      for (Map i in data) {
        productsList.add(ProductsModel.fromJson(i));
      }
      return productsList;
    } else {
      return productsList;
    }
  }

  @override
  Widget build(BuildContext context) {
    return Scaffold(
      appBar: AppBar(
        centerTitle: true,
        title: Text('Api Course'),
```

- The API endpoint is https://webhook.site/94ca76e9-b44a-49f7-8315-e31c31cc8d70. **Sending the Request**: We use **http.get** to send a GET request to the API endpoint. The **Uri.parse** method is used to parse the URL string into a Uri object.
- **Getting the Response**: The **await** keyword is used to wait for the response from the API. The response is stored in the **response variable.**
- **Decoding the Response**: We use jsonDecode to decode the response body into a JSON object. This is because the API returns data in JSON format.
- **Checking the Status Code**: We check if the response status code is 200, which means the request was successful.
- **Parsing the Data**: If the request was successful, we iterate through the JSON data using a for loop. Each iteration, we create a new ProductsModel object using the fromJson method and add it to the productsList.
- **Returning the Data**: Finally, we return the productsList containing the parsed data.

second step =>

the process that the FutureBuilder is doing is called **waiting** or **handling**

the asynchronous data fetching process.



future builder function:

- **Waiting for the data to arrive**: The FutureBuilder is waiting for the data to be fetched from the API and returned as a Future.
- **Handling the data**: Once the data arrives, the FutureBuilder is handling the data by building the widget tree based on the data.
- **Handling errors**: If there's an error during the data fetching process, the FutureBuilder is handling the error by displaying an error message.
- **Displaying a loading indicator**: If the data is still being fetched, the FutureBuilder is displaying a loading indicator to let the user know that the data is being loaded.

what is snapshot

A Snapshot is like a delivery status update from the food delivery service. It tells you whether your food has arrived, or if there's an issue with the delivery. In Flutter, a Snapshot is an object that represents the state of a future at a particular point in time.

```
FutureBuilder(
  future: fetchWeatherData(), // This is like ordering food online
  builder: (context, snapshot) { // This is like getting a delivery status update
    if (snapshot.hasData) { // If the food has arrived
      return Text('The weather is ${snapshot.data}'); // Display the weather data
```

```
} else if (snapshot.hasError) { // If there's an issue with the delivery
 return Text('Error: ${snapshot.error}'); // Display an error message
} else { // If the food is still on its way
 return CircularProgressIndicator(); // Display a loading indicator
```

The Snapshot object has several properties that you can use to determine the state of the future:

- connectionState: This property indicates the state of the future. It can be one of the following values:
  - waiting: The future is still waiting for the data to arrive.
  - done: The future has completed and the data is available.
- hasData: This property indicates whether the future has data available.
- hasError: This property indicates whether the future has an error.
- data: This property contains the actual data returned by the future.