COVID Tracker App:

The app provides real-time insights on COVID-19 statistics, including cases, recoveries, and deaths. Data is fetched from a reliable API and displayed through an interactive user interface. The app features:

- **Country-wise Data Tracking**: Users can select specific countries to view detailed statistics such as daily cases, death counts, and recoveries.
- **Data Visualization**: Statistics are presented using dynamic **map charts**, making it easy to track the spread of the virus globally or within specific regions.
- **Smooth Animations**: Animations are incorporated for chart transitions, screen changes, and data loading to enhance the user experience.
- **Optimized Code**: The app's code is written with performance and efficiency in mind, following best practices and ensuring maintainability.
- **Quality Design Patterns**: The app architecture follows clean and modular design patterns (like **MVVM** or **Provider** for state management), ensuring scalability and ease of updates.

Dependencies Included

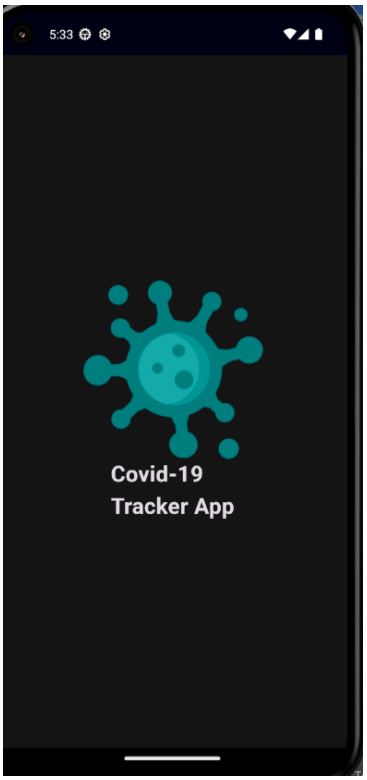
The following dependencies are used in the COVID Tracker App to enhance functionality and design:

- **cupertino_icons: ^1.0.2**
  Provides iOS-style icons.
- **http: ^0.13.4**
  A simple HTTP client to make network requests and fetch COVID-19 data from APIs.
- **animated_text_kit: ^4.2.1**
  Enables animated text, enhancing the app's UI with engaging text effects.
  Documentation
- **pie_chart: ^5.1.0**
  A library for drawing pie charts to visualize COVID-19 data (e.g., cases, deaths, recoveries).
  Documentation
- **flutter_spinkit: ^5.1.0**
  Provides animated loading spinners for a better user experience during data fetching.
  Documentation
- **shimmer: ^2.0.0**
  Creates shimmering loading placeholders while data is being loaded.
  Documentation

BUILDING ANIMATED SPLASH SCREEN

1. **What is a Splash Screen?**

A splash screen is what your users first see when they open your app—it's like a welcome mat! Here, we've created a cool animation where a virus image spins, giving your app a bit of flair before it loads the main content.

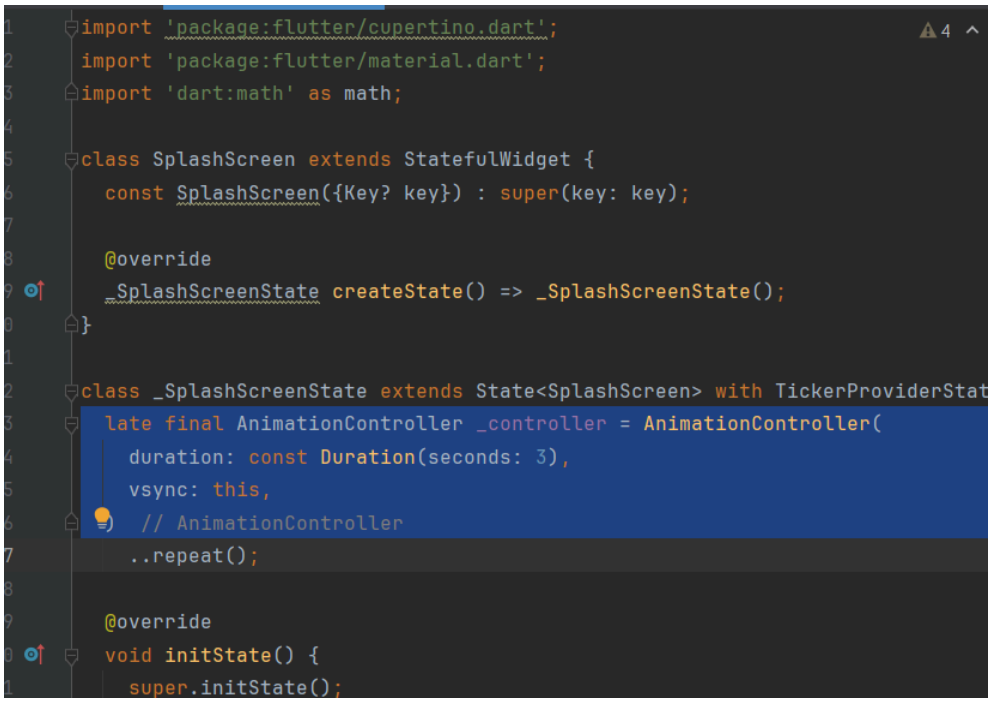


2. **Breaking Down the Code!**

- **Imports:** We've got the usual suspects here—flutter/material.dart for designing the UI and dart:math because we're rotating the virus image using some math!

```dart
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import 'dart:math' as math;
```

- **SplashScreen Class:** The SplashScreen is a **StatefulWidget** because the animation is going to change over time, and we need to track that change.
- **What is the Animation Controller?**
  The AnimationController in Flutter is what powers the movement or changes in your animation. Think of it like a DJ controlling the beats and tempo of the animation. It manages how long the animation should run, how fast it should be, and can even loop the animation, as we've done here.



```dart
import 'package:flutter/cupertino.dart';
import 'package:flutter/material.dart';
import 'dart:math' as math;

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

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

class _SplashScreenState extends State<SplashScreen> with TickerProviderStat
  late final AnimationController _controller = AnimationController(
    duration: const Duration(seconds: 3),
    vsync: this,
    // AnimationController
    ..repeat();

  @override
  void initState() {
    super.initState();
```

- **How the Controller is Set Up**

1. Defining the AnimationController

The AnimationController is declared as **late**, meaning we're telling Flutter that we'll initialize this variable later (in this case, immediately during its setup). This ensures that when it's used, it's already been properly set up.

final means that the controller's reference won't change after it's initialized,

2. Duration of the Animation

The **duration** property controls how long the animation will take. In this case, we set it to 3 seconds. This means that one complete cycle of our animation—rotating the virus image

3. Vsync: Synchronizing the Animation

it keeps the animations in sync with the screen refresh rate. This is important because it helps save resources by ensuring animations only run when they're visible on the screen.

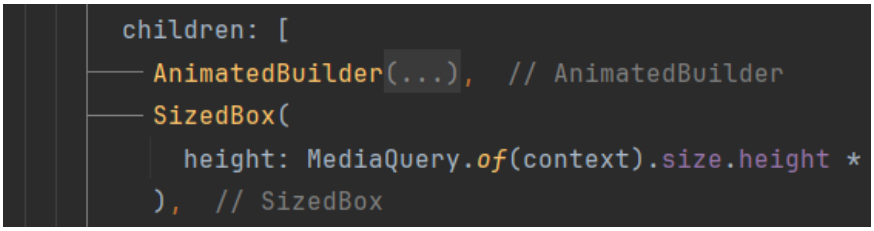
We pass this, which refers to the current state (_SplashScreenState), and since we added the TickerProviderStateMixin, this widget knows how to manage vsync (the ticking clock for the animation).

- **How the Controller Powers the Animation**

The controller's value ranges between 0 and 1 as it progresses over time (from 0% to 100% of the animation). We use this value to rotate the image continuously.

1. **The Role of AnimatedBuilder**
   The AnimatedBuilder widget is the core piece that ties the animation controller to the

widget we want to animate. Think of it like a bridge between the controller (which controls the animation timing) and the widget (which is the thing you're animating).



The **animation** property connects the AnimatedBuilder to our **controller**. It tells the AnimatedBuilder that it should listen for changes in the animation's value (controlled by _controller).

The **child** property is the static part of your widget. This doesn't change when the animation is playing; it stays the same throughout.Here, the child is a Container with a specific size (200x200) and contains the virus image (virus.png).

The **builder** function is the key part of this widget. It gets triggered whenever the animation's value changes. This is where we define how the child (our image) should be transformed or animated.The builder provides the child that we passed earlier and gives us access to the current context.

Transform.rotate is used to rotate the child widget. We rotate it based on the value of the **controller**.

_controller.value: The controller generates values between 0 and 1 as it runs from start to finish. Since we're repeating the animation, this value constantly goes from 0 to 1 and back.

2.0 * math.pi: This is the equivalent of 360 degrees in radians. So, multiplying _controller.value by 2π gives us a full rotation (one complete circle)

```
AnimatedBuilder(
 animation: _controller,
 child: Container(
  height: 200,
  width: 200,
  child: const Center(
   child: Image(image: AssetImage('images/virus.png')),
  ),
 ),
 builder: (BuildContext context, Widget? child) {
  return Transform.rotate(
   angle: _controller.value * 2.0 * math.pi,
   child: child,
  );
 },
),
```

```
SizedBox(
 height: MediaQuery.of(context).size.height * 00,
),
const Align(
  alignment: Alignment.center,
  child: Text('\t\t\tCovid-19\n Tracker App',
    style:
      TextStyle(fontWeight: FontWeight.bold, fontSize: 25)))
```

DESIGNING DISPLAYING PIE CHART

Dummy stats/structuring layout

Why Do We Need TickerProviderStateMixin?

The TickerProviderStateMixin provides a **Ticker** to the AnimationController. This is necessary because without a ticker, the controller wouldn't know when to advance the animation to the next frame.

The Pie Chart

```
PieChart(
 dataMap: {"Total": 20, "Recovered": 15, "Deaths": 5},
 chartRadius: MediaQuery.of(context).size.width / 3.2,
 legendOptions:
   const LegendOptions(legendPosition: LegendPosition.left),
 animationDuration: Duration(milliseconds: 1200),
 chartType: ChartType.ring,
 colorList: colorList,
),
```

dataMap: Here's where we give the data for our chart. Right now, it's fake data—20 total cases, 15 recovered, and 5 deaths.

chartRadius: This is the size of the chart, calculated dynamically based on screen size. It makes the chart responsive.

chartType: ChartType.ring: We are using a **ring** chart, which is essentially a doughnut-style chart.

colorList: These are the colors for each section: Blue for total, green for recovered, and red for deaths.

Reusable Rows

Next up, we've got a **Card** displaying some reusable rows of data. This is going to show various COVID-19 stats, and I've created a custom ReusableRow widget to keep things tidy and reusable.

```
child: Card(
 child: Column(
  children: [
   ReusableRow(title: "Total", value: "200"),
   ReusableRow(title: "Total", value: "200"),
   ReusableRow(title: "Total", value: "200")
  ],
 ),
```

Custom Button

```
 Container(
  height: 50,
  decoration: BoxDecoration(
    color: const Color(0xff1aa260),
    borderRadius: BorderRadius.circular(10)),
  child: const Center(
   child: Text("Track Countries"),
  ),
 )
],
```

ReusableRow Widget

It’s super handy because it lets us display title-value pairs like “Total cases” and the number next to it. Each row has some spacing, a divider underneath, and that’s it!

```
lass ReusableRow extends StatelessWidget {
 String title, value;

 ReusableRow({Key? key, required this.title, required this.value})
   : super(key: key);

 @override
 Widget build(BuildContext context) {
  return SafeArea(
   child: Padding(
    padding: const EdgeInsets.only(left: 10, right: 10, top: 10, bottom: 5),
    child: Column(
     children: [
      Row(
       mainAxisAlignment: MainAxisAlignment.spaceBetween,
       children: [
        Text(title),
        Text(value),
       ],
      ),
      SizedBox(
```

```
          height: 5,
        ),
        Divider(),
      ],
    ),
   ),
  );
 }
}
```

Animation Controller

It makes sure the chart rotates smoothly for 3 seconds. Don't forget to dispose of it in the dispose() method to free up resources:

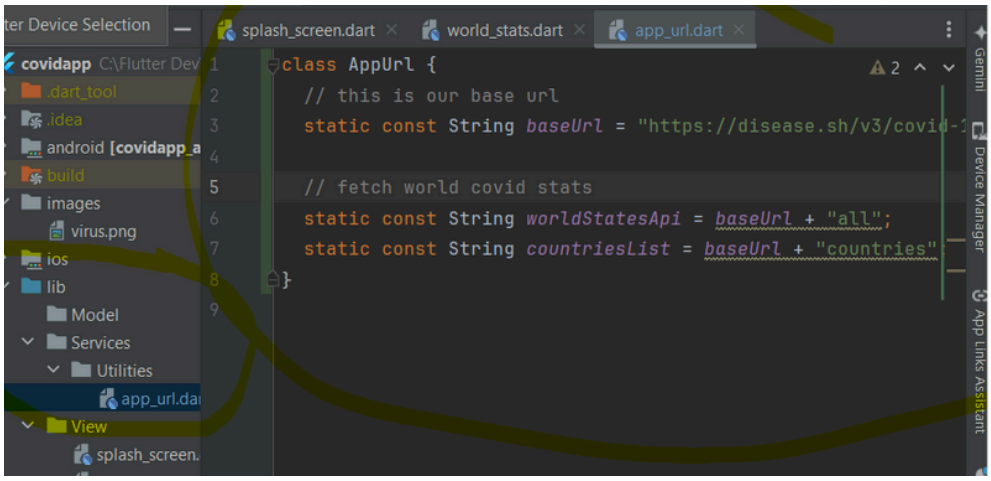
```
class _WorldStatsState extends State<WorldStats> with TickerProviderStateMixin {
 late final AnimationController _controller = AnimationController(
  duration: const Duration(seconds: 3),
  vsync: this,
 )..repeat();
 void  dispose() {
  // TODO: implement dispose
  super.dispose();
  _controller.dispose();
 }
```

Model Directory: Stores data models representing the structure of the data you fetch from APIs (e.g., COVID stats).

Services Directory: Contains functions to make API calls, like fetching global or country-specific COVID stats.

Utilities Directory: Holds reusable utilities and constants, like the app_url.dart file, which manages API URLs.

app_url.dart: Centralizes API URLs (base and endpoints) for easy access and updates, improving maintainability and reusability across the app.

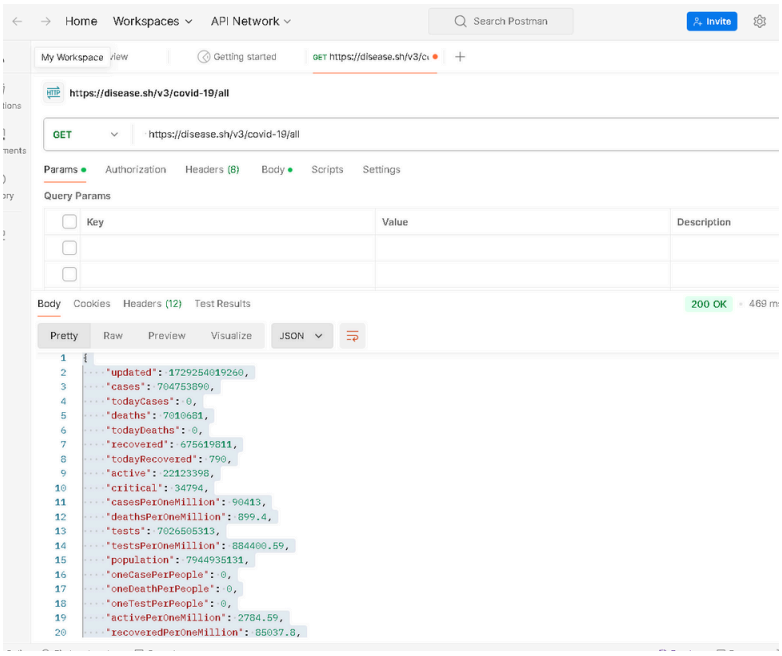


Why is this setup important?

1. **Separation of Concerns**: You're separating your code based on functionality. The app_url.dart file contains only the API URLs, so if you ever need to change the API endpoints or base URL, you only have to update it in one place.
2. **Maintainability**: Keeping different logic in different directories makes your app easier to maintain and extend. If your app scales, you can easily add more APIs or models without cluttering your main files.
3. **Reusability**: By using constants for URLs, you can reference them anywhere in your app without hardcoding them multiple times. This minimizes errors and enhances reusability.

WorldStatsModel Class

we created a world stats model class by simply copying the results fetched from disease api using postman and created a model out of it



below is the explanation of the model class

1. **The Data Model**:

What does a data model do? It's like a blueprint for how we organize the data coming in from an API.

```
class WorldStatsModel {
}
```

It all starts with this class. What does it do? It holds all the different stats we're getting from the API, like total cases, total deaths, tests, and more!The **2. Constructor:**

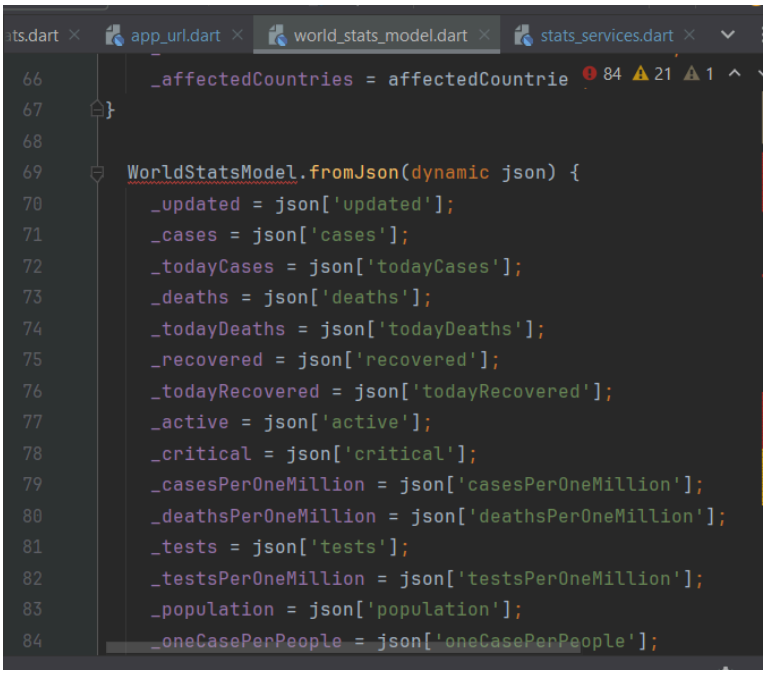
```
WorldStatsModel({
  num updated,
  num cases,
  num todayCases,
  num deaths,
  num todayDeaths,
  num recovered,
  num todayRecovered,
  num active,
  num critical,
  num casesPerOneMillion,
  num deathsPerOneMillion,
  num tests,
  num testsPerOneMillion,
  num population,
  num oneCasePerPeople,
  num oneDeathPerPeople,
  num oneTestPerPeople,
  num activePerOneMillion,
  num recoveredPerOneMillion,
  num criticalPerOneMillion,
  num affectedCountries,})
```

This is our constructor! It's basically the factory that takes all this data and packages it nicely into a WorldStatsModel object.

- Each of these parameters represents a piece of data coming from our COVID API. Think of them like little data boxes: updated is when it was last updated, cases are the total cases, and so on.

3. fromJson() - Mapping JSON Data

we want to be able to get data from the API, right? But the data comes as JSON. So we need to convert that JSON into our model. This is where fromJson comes in!



Getting the Data - fetchWorldStatsRecords()

Now that we have the model ready, let's move on to actually fetching the data from the API using our StatsServices class.

```
 Future<WorldStatsModel> fetchWorldStatsRecords () async{
final response = await http.get(Uri.parse(AppUrl.worldStatsApi));
```

We just sent a GET request to our API to fetch the COVID stats. We're using the http package to talk to the internet.

- http.get: This sends a request to the API URL (in our case, it's stored in AppUrl.worldStatsApi).

```
if (response.statusCode ==200){
 var data = jsonDecode(response.body);
 return WorldStatsModel.fromJson(data);

}else{
 throw Exception('Error');
}
 }
}
```

if the request was successful (status code 200 means success), we grab the JSON data using jsonDecode. Then we convert that JSON into our WorldStatsModel using the fromJson() method we just talked about!

So here's what's happening:

1. Fetch the data from the API.
2. Decode the JSON into a Dart object.

3. Map the JSON data to our model using fromJson.

And finally, if something goes wrong (like, the server is down), we throw an error

Wrap-Up

So, in short, here's what we did:

1. **Created a data model** to structure the API data.
2. **Used fromJson()** to map the incoming JSON to our Dart object.
3. **Fetched the data** from the API and returned it as a nice, clean model.

how to track countries' COVID data in our app! 🦠🌍

Step 1: Setting up the search and list view! 🔍

So, the first thing we've got here is the CountriesList widget, which is a StatefulWidget. Why? Because we want our app to *dynamically* update based on what the user types in the search bar! 🎯

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

 @override
 State<CountriesList> createState() => _CountriesListState();
}

class _CountriesListState extends State<CountriesList> {
 TextEditingController searchController = TextEditingController();
 List<dynamic> countriesList = [];
 List<dynamic> filteredCountries = [];
```

We've got a **TextEditingController** to handle input from the search bar. 🎤 And two lists: **countriesList** to hold all the data fetched from the API, and **filteredCountries** to store what matches the search query! 🚀

Step 2: The cool search functionality 💡

Alright, so here's where the magic happens. We're using a **TextFormField** as our search bar! When the user types something, it filters the list. ✨

```
onChanged: (value) {
    setState(() {
     filteredCountries = countriesList
        .where((country) => country['country']
        .toLowerCase()
        .contains(value.toLowerCase()))
        .toList();
```

```
    });
  },
 ),
),
```

As you type, it checks if the country name contains that value. The setState function refreshes the screen! Boom, instant results! 💥

Step 3: Displaying the countries 🌍

Now, let's display our data. We're using a **FutureBuilder** to fetch the countries' data from the API using our StatsServices class! 📡

So, if we don't have the data yet, we show a loading spinner! ⏳ But once the data is fetched, we assign it to countriesList and initialize filteredCountries with the same data.

```
// FutureBuilder to fetch and display countries list
Expanded(
 child: FutureBuilder(
  future: StatsServices().countriesListApi(), // Calling the API
  builder: (context, AsyncSnapshot<List<dynamic>> snapshot) {
   // Show loading shimmer if no data yet
   if (!snapshot.hasData) {
    return ListView.builder(
     itemCount: 8, // Number of shimmer placeholders
     itemBuilder: (context, index) {
      return Shimmer.fromColors(
       baseColor: Colors.grey.shade700,
       highlightColor: Colors.grey.shade100,
       child: Column(
        children: [
         ListTile(
          title: Container(
           height: 10,
           width: 80,
           color: Colors.white,
          ),
          subtitle: Container(
           height: 10,
           width: 60,
           color: Colors.white,
          ),
          leading: Container(
           height: 50,
           width: 50,
           color: Colors.white,
          ),
```

```
            ),
          ],
        ),
      );
    },
  );
} else if (snapshot.hasError) {
  return Center(
    child: Text('Error: ${snapshot.error}'),
  );
} else {
  // Store fetched data and filter as per search query
  countriesList = snapshot.data!;
  filteredCountries = searchController.text.isEmpty
      ? countriesList
      : filteredCountries;
```

Step 4: Showing each country 📜

Now that we have the data, we're going to show each country in a **ListView**. 🎉

```
return ListView.builder(
 itemCount: filteredCountries.length,
 itemBuilder: (context, index) {
  var countryData = filteredCountries[index]; // Accessing country data
  return Column(
   children: [
    ListTile(
     // Country name
     title: Text(countryData['country']),

     // Cases information
     subtitle: Text('Cases: ${countryData['cases']}'),

     // Display flag image
     leading: SizedBox(
      height: 50,
      width: 50,
      child: Image.network(countryData['countryInfo']['flag']),
     ),

  // On tap, navigate to DetailScreen with the countryData
     onTap: () {
      Navigator.push(
       context,
```

```
          MaterialPageRoute(
            builder: (context) => DetailScreen(countryData: countryData),
          ),
        );
      },
    ),
    const Divider(), // Divider between list items
  ],
 );
},
);
```

Each item in the list shows the **country name** and the **flag**! 🇺🇳 And when you tap on a country, it takes you to the detailed screen with more info about that country's COVID stats.

Step 5: Fetching data from the API 🌐

Finally, we have the StatsServices class. This is the backend magic that grabs the countries' COVID data from an API. 🧙‍♂️

```
// Fetch Countries List
Future<List<dynamic>> countriesListApi() async {

  final response = await http.get(Uri.parse(AppUrl.countriesList));

  if (response.statusCode == 200) {
   var data = jsonDecode(response.body);
   return data;
  } else {
   throw Exception('Error: Failed to load countries list');
  }

}

}
```

So, it's making an **HTTP GET request** to the API, and once it gets the data, it decodes the JSON and passes it to our app. 🚀 If the request fails, we throw an error, but you can easily handle that by showing an error message!