Chapter 3: Displaying Data from REST APIs with Retrofit

In this chapter, we'll be taking a break from the Jetpack libraries and focusing on adding real data within our Restaurants application by using a very popular networking library on Android called **Retrofit**.

Retrofit is an HTTP client library that lets you create an HTTP client declaratively and abstracts most of the underlying complexity associated with handling network requests and responses. This library allows us to connect to a real web API and retrieve real data within our app.

In the *Understanding how apps communicate with remote servers* section, we will focus on exploring how mobile applications retrieve and send data to remote web APIs. In the *Creating and populating your database with Firebase* section, we will create a database for our Restaurants application with the help of Firebase and fill it with JSON content.

In the *Exploring Retrofit as an HTTP networking client for Android* section, we will learn what Retrofit is, and how it can help us create network requests within our Restaurants app.

Lastly, in the *Improving the way our app handles network requests* section, we will tackle some common issues that occur when Android applications create async work to retrieve data from web APIs. We will identify those issues and fix them.

To summarize, in this chapter, we're going to cover the following main topics:

- Understanding how apps communicate with remote servers
- Creating and populating your database with Firebase

- Exploring Retrofit as an HTTP networking client for Android
- Improving the way our app handles network requests

Before jumping in, let's set up the technical requirements for this chapter.

Technical requirements

Building Compose-based Android projects with Retrofit usually requires just your day-to-day tools. However, to follow along smoothly, make sure you have the following:

- The Arctic Fox 2020.3.1 version of Android Studio. You can also use a newer Android Studio version or even Canary builds but note that IDE interface and other generated code files might differ from the ones used throughout this book.
- Kotlin 1.6.10 or newer installed in Android Studio.
- The Restaurants app code from the previous chapter.
- A Google account to create a Firebase project.

The starting point for this chapter is the Restaurants application that we developed in the previous chapter. If you haven't followed the coding steps from the previous chapter, access the starting point for this chapter by navigating to the **`Chapter_02`** directory of this book's GitHub repository and importing the Android project entitled **`chapter_2_restaurants_app`**.

To access the solution code for this chapter, navigate to the **`Chapter_03`** directory:

https://github.com/PacktPublishing/Kickstart-Modern-Android-Development-with-Jetpack-and-Kotlin/tree/main/Chapter_03.

Understanding how apps communicate with remote servers

Modern applications need to show real content that can change over time and need to avoid hardcoding data, as we did in the previous chapters. Let's briefly cover how they do that.

Most network-connected apps use the HTTP protocol to send or receive data in the format of JSON from REST web services through a REST API.

That's a lot of words we've just thrown at you, so let's break them down:

- **Hypertext Transfer Protocol** (**HTTP**) is a protocol for asynchronously fetching various resources from web servers. In our case, the resources are the data that our application needs to display.
- **JavaScript Object Notation** (**JSON**) is the data format of the content that's transferred in HTTP requests. It's structured, lightweight, and human-readable as it consists of key-value pairs that are easy to parse and commonly used as a suitable format for data exchange between apps and web servers. In our app, we will receive the data from the web server in such JSON format.
- **REST web services** are those sources that contain the requested data and that conform to the **representational state transfer** (**REST**) architecture. REST means that the web server uses the HTTP protocol to communicate resources and that its resources are manipulated with the common HTTP methods: **`GET`**, **`PUT`**, **`POST`**, **`DELETE`**, and so on.
- A **REST API** is an **application programming interface** (**API**) that conforms to the constraints of the REST architecture and allows you to interact with REST web services. The REST API is the contract and the entry point that's used by apps to obtain or send data to and from the backend.

Let's try to visualize the relationship between these entities:

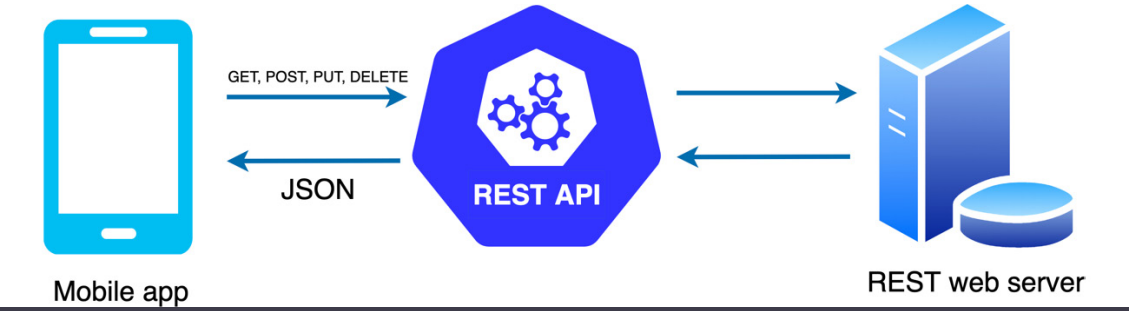

Figure 3.1 – Overview of HTTP communication between apps and web servers

We want to implement something similar for our Restaurants application. For this to work, we will need a REST server. For the sake of simplicity, we will use the Firebase Realtime Database and create a database.

Creating and populating your database with Firebase

So far, we've only used hardcoded data as the source of content for our Restaurants app. Since almost every real application uses dynamic data that comes from a backend server through a REST API, it's time to step up our game and create a database that simulates such a remote API.

We can do this for free with the help of Firebase. Firebase is backed by Google and represents a **Backend-as-a-Service** (**BaaS**), which allows us to build a database very easily. We will use the Realtime Database service from Firebase without using the Firebase Android SDK. Even though such a database is not a proper REST web service, we can use its database URL as a REST endpoint and pretend that that is our REST interface, therefore simulating a real backend.

NOTE

As we mentioned in the Technical requirements section, make sure that you have an existing Google account or that you create one beforehand.

Let's start creating a database:

1. Navigate to the Firebase console and log into your Google account by going to https://console.firebase.google.com/.
2. Create a new Firebase project:

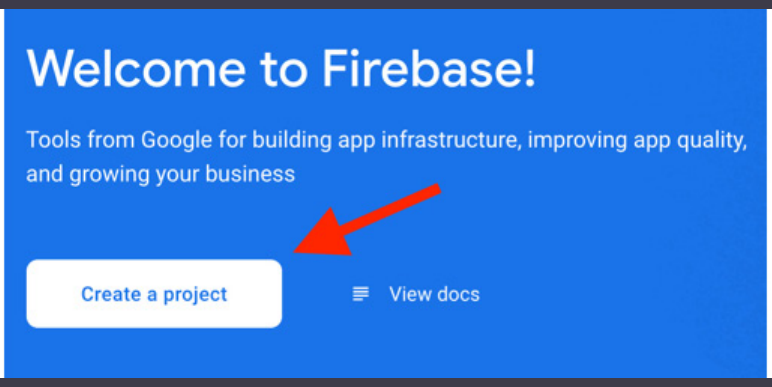

Figure 3.2 – Creating a new Firebase project

3. Input the name of your project (it should be about restaurants!) and press **Continue**.
4. Optionally, in the next dialog, you can opt out from Google Analytics since we won't be using the Firebase SDK. Press **Continue** again. At this point, the project should be created.
5. From the left menu, expand the **Build** tab, search for **Realtime Database**, and then select it:

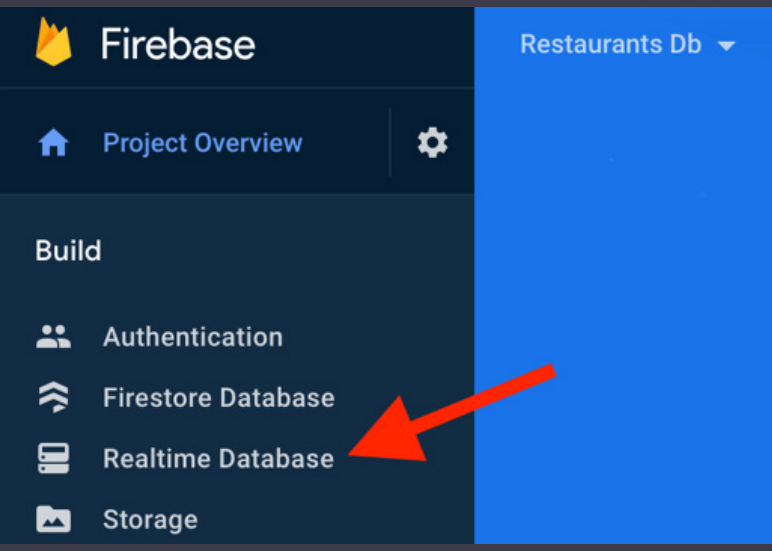

Figure 3.3 – Accessing Realtime Database

6. On the newly displayed page, create a new database by clicking **Create Database**.
7. In the **Set up database** dialog, select a location for your database and then click **Next**:

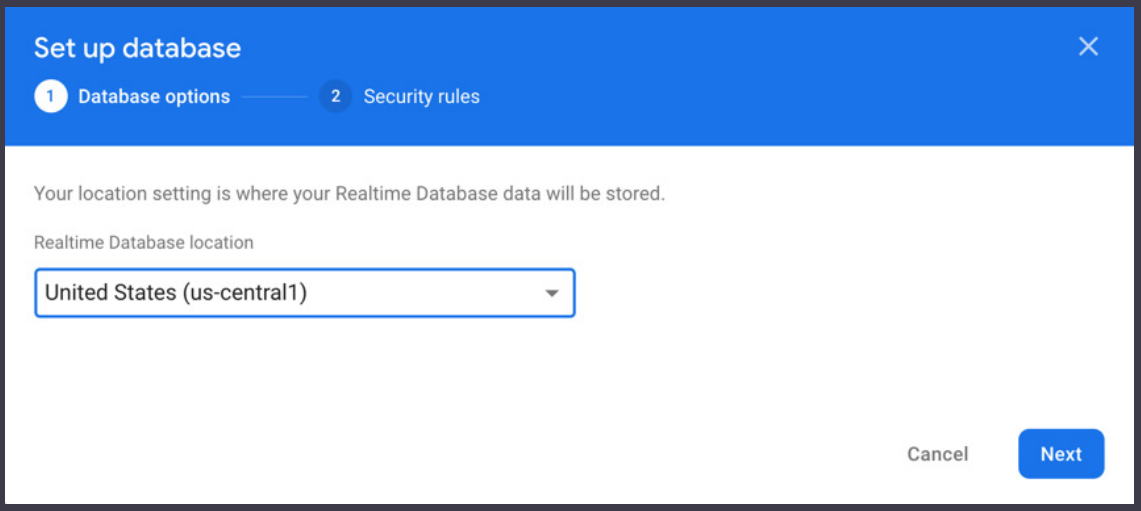

Figure 3.4 – Setting up a Realtime Database

NOTE

If later on any network calls to your Firebase Database fail for no apparent reason, you might find yourself in a Firebase restricted location – as I am writing this chapter, because of the current situation caused by the eastern war, all internet providers from Romania are restricted and any network calls to Firebase Database are failing. If this happens to you, try selecting a different location for your Realtime Database instance.

8. In the same dialog, define your security rules by selecting **Start in test mode** and then clicking **Enable**.

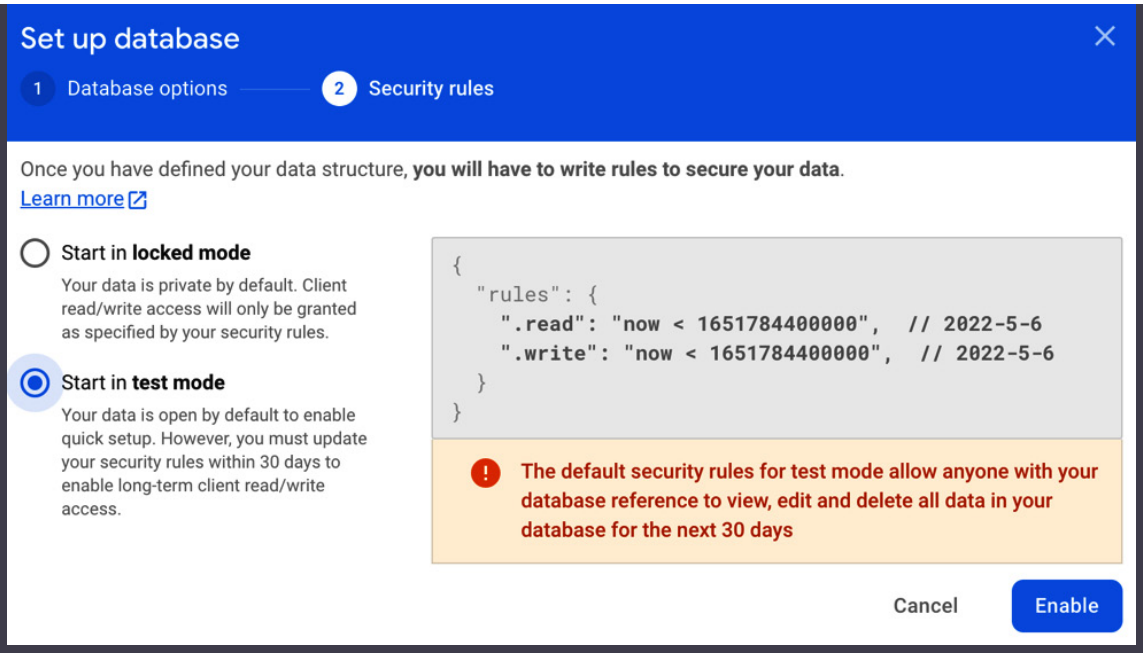

Figure 3.5 – Setting up the security rules of your database

IMPORTANT NOTE

The default security rules for test mode allow anyone to view or modify the content within your database for the next 30 days since creation. After these 30 days, if you want to keep using the database in test mode, you will need to update the security rules by changing the timestamp values for the `".read"` *and the* `".write"` *fields with greater timestamp values. To skip this, we will just set the* `".read"` *and the* `".write"` *fields to* `true` *in the next steps. However, Firebase might still restrict your access if you leave the database open for access without any rules indefinitely – that's why I recommend you visit the Firebase console and check the security rules for your database often to make sure that access was not revoked.*

At this point, you should be redirected to your database's main page in the **Data** tab:

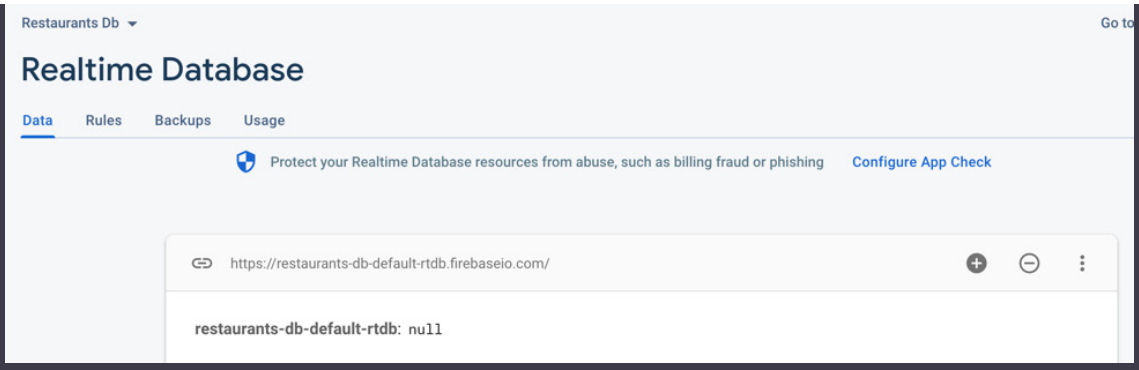

Figure 3.6 – Observing the Realtime Database main page

You will now notice your URL for this database: https://restaurants-db-default-rtdb.firebaseio.com/.

Your URL should be similar but may differ, depending on the name you have chosen for your database.

Note that the database seems to be empty; we only have an empty root node being called after our database: **`restaurants-db-default-rtdb`**. It's time to add data to our database.

9. Access the solution code for this chapter by navigating to the **`Chapter_03`** directory of this book's GitHub repository. Then, select the **`restaurants.json`** file. You can also access it by following this link: https://github.com/PacktPublishing/Kickstart-Modern-Android-Development-with-Jetpack-and-Kotlin/blob/main/Chapter_03/restaurants.json.

From here, download the **`restaurants.json`** file as we will need it shortly. To do that, press on the **Raw** button provided by the Github website and then right click the document that has been opened and download the JSON file by selecting **Saves As**.

10. Go back to the Firebase console, press on the three-dots menu to the right of the database URL, and select **Import JSON**:

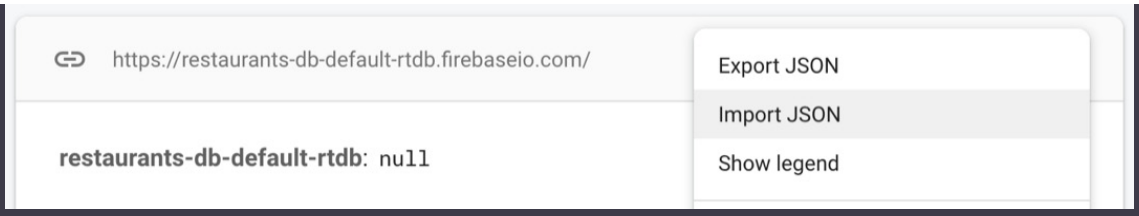

Figure 3.7 – Importing JSON content into Realtime Database

Make sure that you select the **`restaurants.json`** file that you've previously downloaded from the book's GitHub repository.

11. Wait for the page to refresh and check out the content that is populated in the database:

Figure 3.8 – Observing the content's structure in our database

Here, we can see that our database contains a list of **restaurants**. Each restaurant has attributes that are similar to the ones in our **`Restaurant`** class: an ID, title, and description. The restaurants in our database also contain other fields that we will not need right now, so let's ignore them.

NOTE

If you compare the structure of the content in our database with the one from the JSON file we've uploaded, we can see that it is very similar: we have a **`restaurants`** *node that contains an array of objects, each containing consistent key-value pairs. The only exception is the presence of the indexes*

(0, 1, 2, and so on) for each restaurant, which were automatically created by Firebase. We should ignore these as they won't affect us.

Now, even though we set the security rules to **Test Mode** previously, let's revisit them.

12. Move away from the **Data** tab and select the **Rules** tab. Here, to make sure that we can always read data from this database, change the **`".read"`** key's value to **`"true"`**:

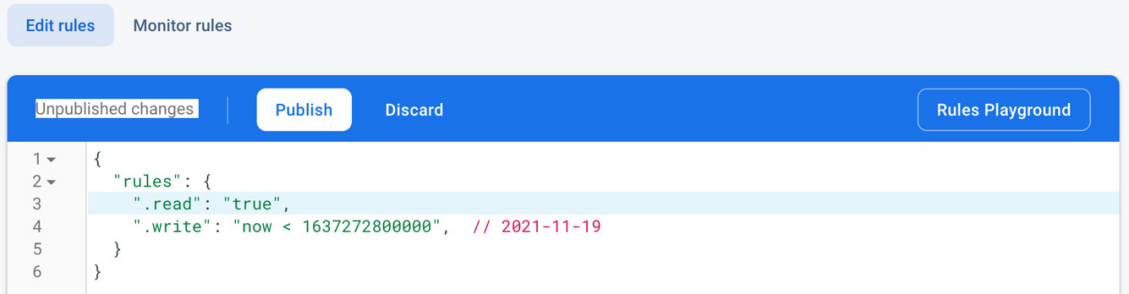

Figure 3.9 – Updating security rules in Realtime Database

13. Press **Publish** to save the changes.

    NOTE

    The security rules that we've set should not be used in a production application. We are hardcoding our security rules to **`true`** *since we are testing and not publishing anything to production.*

Now, we can access the database URL as a simple REST endpoint, simulating a real REST server that we can connect to. To experiment, copy the URL of your newly created database, append **`restaurants.json`**, and paste it into your browser.

Accessing this URL should return the JSON response of our restaurants, whose structure we will cover shortly. Until then, we need to instruct our application to create HTTP requests to obtain that data from our newly created database. So, let's do that next.

Exploring Retrofit as an HTTP networking client for Android

For the application to obtain data from our database, we need to implement an HTTP client that will send network requests to the REST API of the database.

Instead of working with the HTTP library provided by default by Android, we will use the **Retrofit** HTTP client library, which lets you create an HTTP client that is very easy to work with.

If you plan to develop an HTTP client that interfaces with a REST API, you will have to take care of a lot of things – from making connections, retrying failed requests, or caching to response parsing and error handling. Retrofit saves you development time and potential headaches as it abstracts most of the underlying complexity associated with handling network requests and responses.

In this section, we will cover the following topics:

- Using Retrofit
- Adding Retrofit to the Restaurants application
- Mapping JSON to model classes
- Executing GET requests to the Firebase REST API

Let's start with some basics about Retrofit!

Using Retrofit

Retrofit makes networking simple in Android apps. It allows us to consume web services easily, create network requests, and receive responses while reducing the boilerplate code that's usually associated with their implementation.

NOTE

Retrofit also allows you to easily add custom headers and request types, file uploads, mocking responses, and more.

To execute network requests with Retrofit, we need the following three components:

- An interface that defines the HTTP operations that need to be performed. Such an interface can specify request types such as **`GET`**, **`PUT`**, **`POST`**, **`DELETE`**, and so on.
- A **`Retrofit.Builder`** instance that creates a concrete implementation of the interface we defined previously. The Builder API allows us to define networking parameters such as the HTTP client type, the URL endpoint for the HTTP operations, the converter to deserialize the JSON responses, and so on.
- Model classes that allow Retrofit to know how to map the deserialized JSON objects to regular data classes.

Enough with the theory – let's try to implement Retrofit and use the components we introduced previously in our Restaurants application.

Adding Retrofit to the Restaurants application

We want to connect our Restaurants application to the newly created Firebase database and send HTTP network requests to it. More specifically, when the Restaurants application is launched and the **`RestaurantsScreen`** composable is composed, we want to get the list of restaurants at runtime and not depend on hardcoded content within the app.

Let's do that with the help of Retrofit:

1. Inside the **`build.gradle`** file in the app module, add the dependency for Retrofit inside the **`dependencies`** block:

```
implementation
"com.squareup.retrofit2:retrofit:2.9.0"
```

2. After updating the **`build.gradle`** file, make sure to sync your project with its Gradle files. You can do that by clicking on the **File** menu option and then by selecting **Sync Project with Gradle Files**.
3. Create an interface that defines the HTTP operations that are executed between our app and the database. Do so by clicking on the application package, selecting **New**, and then selecting **Kotlin Class/File**. Enter **`RestaurantsApiService`** as the name and select **Interface** as the type. Inside the newly created file, add the following code:

```
import retrofit2.Call
import retrofit2.http.GET
interface RestaurantsApiService {
    @GET("restaurants.json")
    fun getRestaurants(): Call<Any>
}
```

Let's break down the code we've just added into meaningful actions:

- Retrofit turns your HTTP API into a simple Java/Kotlin interface, so we've created a **`RestaurantsApiService`** interface that defines the HTTP actions we need.
- We've defined a **`getRestaurants`** method inside the interface that returns a **`Call`** object with an undefined response type marked by Kotlin's **`Any type`**. Each **`Call`** from **`RestaurantsApiService`** can make a synchronous or asynchronous HTTP request to the remote web server.
- We've annotated the **`getRestaurants`** method with the **`@GET`** annotation, thereby telling Retrofit that this method should execute a **`GET`** HTTP action to obtain data from our web server. Inside the **`@GET`** annotation, we passed the endpoint's path, which represents the **`restaurants`** node within our Firebase database. This means that when we execute this request, the **`restaurants.json`** endpoint will be appended to the base URL of the HTTP client.

NOTE

We mentioned that we can use our Firebase Realtime Database URL as a REST API. To access a specific node, such as the **`restaurants`** *node of our database, we also appended the* **`.json`** *format to make sure that the Firebase database will behave like a REST API and return a JSON response.*

Later on, after we instantiate a Retrofit builder, the library will know how to turn our **`getRestaurants`** method into a proper HTTP call.

But before that, you have probably noticed that the **`getRestaurants`** HTTP request from our interface has its response type defined as **`Any`**. We expect to receive the JSON content of our restaurants mapped to data classes that we can use in our code. So, let's work on that next.

Mapping JSON to model classes

Retrofit lets you automatically serialize request bodies and deserialize response bodies. In our case, we're interested in deserializing the response body, from JSON into Java/Kotlin objects.

To deserialize the JSON response, we will instruct Retrofit to use the GSON deserialization library, but until then, let's have a look at the JSON response that our database returns. Remember that we imported a JSON file called **`restaurants.json`** when we populated the Firebase database.

Let's open that file with any text editor and observe its structure:

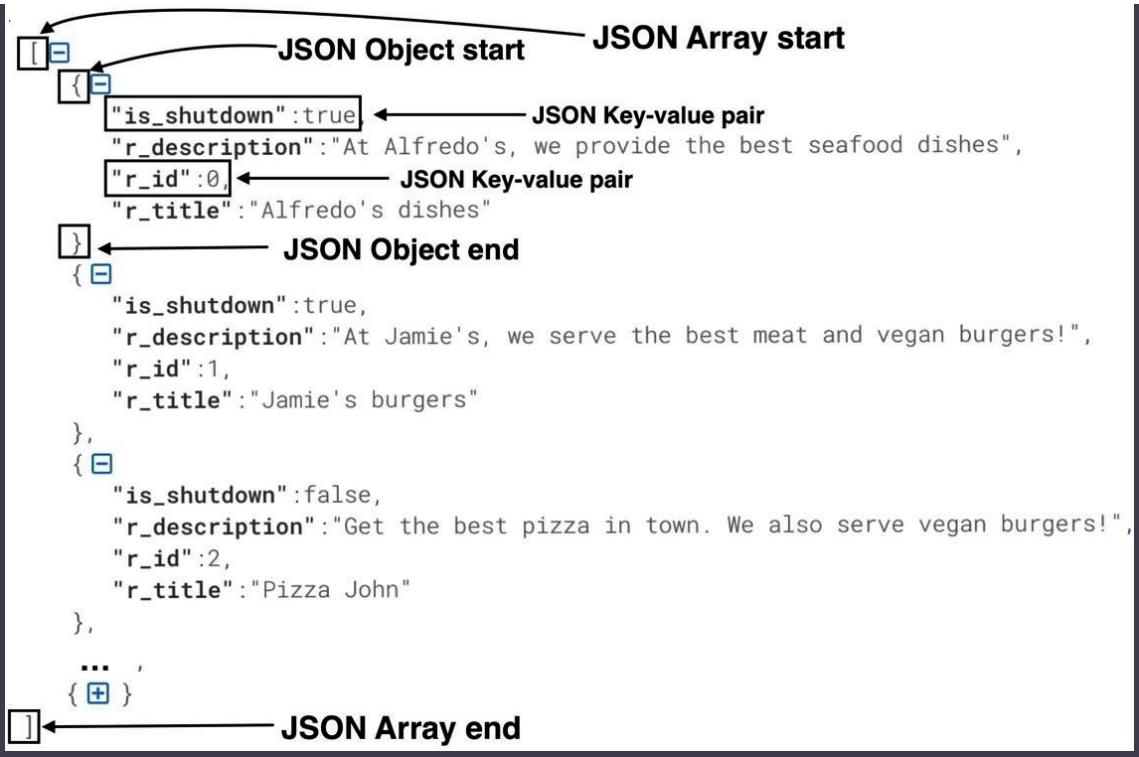

Figure 3.10 – The JSON structure of the Firebase database's content

We can observe the following elements in the JSON response structure:

- It contains an array of JSON objects marked by the `[` and `]` identifiers.
- The contents of the JSON array's element are marked by the `{` and `}` identifiers and they enclose the JSON object structure of a restaurant.
- The restaurant JSON object contains four key-value pairs, separated by the `,` separator.

The response from our database will be of type `List<?>` since the response holds an array of JSON objects. The main question that remains is, what data type should our application expect inside that list?

To answer that, we must inspect the structure of the restaurant JSON object a bit closer:

```
{
   "is_shutdown":false,
   "r_description":"Get the best pizza in town. We also serve vegan burgers!",
   "r_id":2,
   "r_title":"Pizza John"
},
```

Figure 3.11 – The structure of the restaurant JSON object

Here, we can see that the JSON restaurant has four key-value pairs that refer to the **id**, **title**, **description**, and **shutdown** statuses of the corresponding restaurant. This structure is similar to our **Restaurant.kt** data class within the project:

```
data class Restaurant(val id: Int,
                      val title: String,
                      val description: String,
                      var isFavorite: Boolean =
false)
```

Our **Restaurant** also contains the **id**, **title**, and **description** fields. We are not interested in the **shutdown** status for now, so it's tempting to use the **Restaurant** class as the model for our response, thus making our **getRestaurants()** method in **RestaurantsApiService** return **List<Restaurant>** as the response of the request.

The issue with this approach is that we need to tell Retrofit to match the **r_id** key's value with our **id** field. The same goes for **r_title**, which should be matched with the **title** field and so on. We can approach this in two ways:

- Rename the **Restaurant** data class fields so that they match the response keys: **r_id**, **r_title**, and so on. In this case, the deserialization will automatically match our fields with the fields from the JSON objects since the JSON keys are identical to the fields' names.
- Annotate the **Restaurant** data class fields with special serialization matchers that tell Retrofit which keys should be matched with each field. This won't change the variable names.

The first approach is bad because our **Restaurant** data class would end up with fields that contain underscore naming dictated by the server. It would also not comply with Kotlin's CamelCase guideline for defining field variables anymore.

Let's choose the second approach, where we specify the serialization keys ourselves. To do that, we will tell Retrofit to deserialize the JSON with the GSON deserialization library, which is a powerful framework for converting JSON strings into Java/Kotlin objects and vice versa:

1. First, we need to add the GSON library dependency to mark our fields with custom serialization keys. Inside the **`build.gradle`** file in the app module, add the dependency for GSON inside the **`dependencies`** block:
```
implementation "com.google.code.gson:gson:2.8.6"
```
2. After updating the **`build.gradle`** file, make sure to sync your project with its Gradle files. You can do that by clicking on the **File** menu option and then by selecting **Sync Project with Gradle Files**.
3. Inside **`Restaurant.kt`**, add the **`@SerializedName`** annotation for each field and specify the corresponding serialization keys from the JSON structure:
```
import com.google.gson.annotations.SerializedName
data class Restaurant(
    @SerializedName("r_id")
    val id: Int,
    @SerializedName("r_title")
    val title: String,
    @SerializedName("r_description")
    val description: String,
    var isFavorite: Boolean = false)
```

By doing so, we've made sure that Retrofit will correctly match each of the JSON key's values with our corresponding field inside the **`Restaurant`** data class while also matching the data type:

- The **`r_id`** key matches the **`id`** field. The **`r_id`** key has a whole number as its value in the JSON structure, so we stored this key's value in the **`id: Int`** field.
- The **`r_title`** key matches the **`title`** field. The **`r_title`** key has text as its value marked with the **`"`** and **`"`** identifiers, so we stored this key's value in the **`title: String`** field.

- The **`r_description`** key matches the **`description`** field. The **`r_description`** key has text as its value marked with the **`"`** and **`"`** identifiers, so we stored this key's value in the **`description: String`** field.

NOTE

For now, we are using the Restaurant data model both as the API response model and as the domain model that's used throughout the application. Architecturally, this practice is not recommended, and we will cover why this is the case and fix it in Chapter 8, Getting Started with Clean Architecture in Android.

4. Update the **`getRestaurants()`** method inside **`RestaurantsApiService`** so that it returns a **`Call`** object from the server with the type parameter that matches the response that is expected. In our case, that would be **`List<Restaurant>`**:

```
interface RestaurantsApiService {
    @GET("restaurants.json")
    fun getRestaurants(): Call<List<Restaurant>>
}
```

With that, our Retrofit API interface has been defined to receive the content of the Restaurants database from our Firebase database. The only step left is to configure a Retrofit builder instance and execute the request.

Executing GET requests to the Firebase REST API

Let's configure the last component that's needed to perform requests with Retrofit – the **`Retrofit.builder`** object:

1. First, we need to add the GSON Converter library dependency for Retrofit so that Retrofit deserializes the JSON response while following the GSON serialization annotations we added previously. Inside the

build.gradle file in the app module, add the dependency for the Retrofit GSON converter inside the **dependencies** block:

```
implementation "com.squareup.retrofit2:converter-
    gson:2.9.0"
```

2. After updating the **build.gradle** file, make sure to sync your project with its Gradle files. You can do that by clicking on the **File** menu option and then by selecting **Sync Project with Gradle Files**.
3. Inside **RestaurantsViewModel**, add a **restInterface** variable of type **RestaurantsApiService** and create an **init** block where we will instantiate the **Retrofit.builder** object:

```
class RestaurantsViewModel(…) : ViewModel() {
    private var restInterface:
RestaurantsApiService
    val state = mutableStateOf(
        dummyRestaurants.restoreSelections()
    )
    init {
        val retrofit: Retrofit = Retrofit.Builder()
            .addConverterFactory(
                GsonConverterFactory.create()
            )
            .baseUrl(
                "https://restaurants-db-default
                        -rtdb.firebaseio.com/"
            )
            .build()
        restInterface = retrofit.create(
            RestaurantsApiService::class.java
        )
    }
    […]
}
```

We've added all the necessary pieces for our networking client. Let's break this code down:

- First, we've defined a **`restInterface`** variable of type **`RestaurantsApiService`** that we will call upon to execute the desired network requests. At this point, the **`restInterface`** variable holds no value.
- We've added an **`init`** block to instantiate the Retrofit builder object. As the primary constructor can't contain any code, we are placing the initialization code in an initializer block prefixed with the **`init`** keyword.
- We've instantiated a **`retrofit: Retrofit`** variable with the **`Retrofit.Builder`** accessor and specified the following:
- A **`GsonConverterFactory`** to explicitly tell Retrofit that we want the JSON to be deserialized with the GSON converter, following the **`@Serialized`** annotations we specified in the **`Restaurant`** data class.
- A **`baseUrl`** for all the requests that are to be executed – in your case, replace this URL with the URL of your Firebase database.
- Finally, we called **`.create()`** on the previously obtained **`Retrofit`** object and passed our interface with the desired requests: **`RestaurantsApiService`**. Behind the scenes, Retrofit creates a concrete implementation of our interface that will handle all the networking logic, without us having to worry about it. We store this instance from Retrofit inside our **`restInterface`** variable.

Now, we can execute requests – in our case, the request to get the list of restaurants.

4. Inside **`RestaurantsViewModel`**, add the **`getRestaurants`** method:

```
fun getRestaurants() {
    restInterface.getRestaurants().execute().body()
        ?.let { restaurants ->
            state.value =
restaurants.restoreSelections()
        }
}
```

We've added all the necessary steps for our networking request to be executed. Let's break this code down:

1. We've obtained a **`Call`** object called **`Call<List<Restaurant>>`** from our Retrofit **`restInterface`** variable by calling the **`getRestaurants()`** interface method. The **`Call`** object represents the invocation of a Retrofit method that sends network requests and receives a response. The type parameter of the **`Call`** object matches the response type; that is, **`<List<Restaurant>>`**.
2. On the previously obtained **`Call`** object, we called **`execute()`**. The **`execute()`** method is the most simple approach to starting a network request with Retrofit as it runs the request synchronously on the main thread (the UI thread) and blocks it until the response arrives. No network request should block the UI thread yet, though we will fix this soon.
3. The **`execute()`** method returns a Retrofit **`Response`** object that allows us to see if the response was successful and obtain the resulting body.
4. The **`body()`** accessor returns a nullable list of type **`List<Restaurant>>?`**. We apply the Kotlin **`let`** extension function and name the list **`restaurants`**.
5. We pass the resulting **`restaurants`** list to our **`state`** object after restoring the selections in case of system-initiated process death, similar to what we did for the initial state value.

With that, we've instructed our **`ViewModel`** on how to obtain the list of restaurants from the database and to pass this result to our screen's state. One issue that we will have to address later is that we are not catching any errors that may be thrown by Retrofit if the request fails. Until then, let's focus on updating the state with the new result.

5. Inside **`RestaurantsViewModel`**, we need to update the state's initial value so that it contains an empty list. This is because, when the screen is first displayed, we no longer have restaurants to render – we will get them later in the network request. Update the initial value of the **`state`** object by removing **`dummyList`** and placing an **`emptyList()`** instead:

```
  val state = mutableStateOf(emptyList<Restaurant>())
```

6. Inside the **`Restaurant.kt`** file, remove the **`dummyRestaurants`** list since we will be obtaining the restaurants at runtime through the previously defined request.
7. We want to trigger the network request to obtain the restaurants from the server. Inside **`RestaurantsScreen.kt`**, update the **`RestaurantsScreen`** composable function so that it calls the **`getRestaurants()`** method of **`viewModel`**, which will trigger the network request to obtain the restaurants from the server:

```
@Composable
fun RestaurantsScreen() {
    val viewModel: RestaurantsViewModel =
viewModel()
    viewModel.getRestaurants()
    LazyColumn( … ) { … }
}
```

By calling **`viewModel.getRestaurants()`**, we are trying to load the list of restaurants when the **`RestaurantsScreen`** composable is composed for the first time. This practice is not recommended and we will see in the following steps why that is and how we can fix it.

8. Add internet permission inside the **`AndroidManifest.xml`** file:

```
<manifest xmlns:android="…"
    package="com.codingtroops.restaurantsapp">
    <uses-
permission
android:name="android.permission.INTERNET" />
      <application> … </application>
</manifest>
```

9. Run the application by clicking the **Run** button.

Unfortunately, the application will most likely crash. If we check **Logcat**, we will notice an exception stack similar to the following:

```
2021-10-27 10:38:47.918 23484-23484/com.codingtroops.restaurantsapp E/AndroidRuntime: FATAL EXCEPTION: main
    Process: com.codingtroops.restaurantsapp, PID: 23484
    android.os.NetworkOnMainThreadException
```

Figure 3.12 – Crash stack trace for executing a network request on the main thread

The exception that's been thrown here is a **`NetworkOnMainThreadException`** and it's clear what's wrong with our code: we are executing a network request on the Main thread.

This has happened because with the Android Honeycomb SDK, *executing network requests on the Main thread is forbidden* because the UI of the application will freeze until the response from the server arrives, making the app unusable in that timeframe. In other words, we can't and shouldn't use the **`.execute()`** method of a Retrofit **`Call`** object because the request will run synchronously on the Main thread.

Instead, we can use an alternative that not only will execute the requests asynchronously and on a separate thread, but also allow us to handle any errors that are thrown by Retrofit.

10. In the **`getRestaurants()`** method of **`ViewModel`**, replace the **`.execute()`** call with **`.enqueue()`**:

```
fun getRestaurants() {
    restInterface.getRestaurants().enqueue(
     object : Callback<List<Restaurant>> {
         override fun onResponse(
             call: Call<List<Restaurant>>,
             response: Response<List<Restaurant>>
         ) {
             response.body()?.let { restaurants ->
                 state.value =
                     restaurants.restoreSelections()
             }
         }
         override fun onFailure(
```

```
            call: Call<List<Restaurant>>, t:
Throwable
        ) {
            t.printStackTrace()
        }
    })
}
```

When adding the missing imports for the **`Call`**, **`Callback`** and **`Response`** classes, make sure that you're adding the Retrofit2 imports that start off like this: **`import retrofit2.*`**.

Getting back to the code that we've added, let's look at it in more detail:

- On the **`Call`** object that we obtained from our **`restInterface.getRestaurants()`** method, we called the **`.enqueue()`** function. The **`enqueue()`** call is a better alternative to **`.execute()`** since it runs the network request asynchronously on a separate thread, so it will no longer run on the UI thread and it won't block the UI.
- The **`.enqueue()`** function receives a **`Callback`** object as an argument that allows us to listen for success or failure callbacks. The **`Callback`** object's type parameter defines the expected **`Response`** object. Since we expect a response of type **`<List<Restaurant>>`**, the returned **`Callback`** type is defined as **`Callback<List<Restaurant>>`**.
- We've implemented the required **`object : Callback<List<Restaurant>>`** and implemented its two callbacks:
- **`onResponse()`**, which is the success callback that's invoked when the network request succeeds. It provides us with the initial **`Call`** object, but more importantly the **`Response`** object; that is, **`Response<List<Restaurant>>`**. Inside this callback, we get the body from the response and update the value of the **`state`** variable, just like we did with **`execute()`**.
- **`onFailure()`**, which is the failure callback. It's invoked when a network exception occurs while talking to the server or when an unexpected exception occurs while creating the request or processing the re-

sponse. This callback provides us with the initial **`Call`** object and the **`Throwable`** exception that was intercepted and whose stack trace we print.

Now, you can run the application. It shouldn't crash anymore since calling **`enqueue()`** allowed the request to run on a separate thread so that we could safely wait for the response without blocking the UI.

NOTE

As a good practice, make sure that when you're making requests with Retrofit, you always call the **`enqueue()`** *function and not* **`execute()`**. *You want your users to not experience crashes and to be able to interact with the app while they're waiting for the network response.*

Yet even with this addition, there are still two concerning issues with our code. Were you able to notice them? Let's try to identify them.

Improving the way our app handles network requests

Our application now successfully obtains data from the server dynamically, at runtime. Unfortunately, we have made two major mistakes in our code, and both are related to how the app handles the requests. Let's identify them:

- First, we are not canceling our network request as a cleanup measure. If our UI component that is bound to **`RestaurantsViewModel`** – in our case, **`MainActivity`** – is destroyed before the response from the server can arrive (for example, if the user navigates to another activity), we could potentially create a memory leak. This is because our **`RestaurantsViewModel`** would still be tied to the **`Callback<List<Restaurant>>`** object, which waits for the server's re-

sponse. Due to this, the garbage collector won't free up the memory associated with both of their instances.

- Secondly, we are not triggering the network request from a controlled environment. The **`viewModel.getRestaurants()`** method is called inside the **`RestaurantsScreen()`** composable function without any special considerations. This means that every time the UI is recomposed, the composable will ask **`ViewModel`** to execute network requests, resulting in possible multiple and redundant requests.

Let's focus on the first issue for now.

Canceling network requests as a cleanup measure

The main problem in our **`RestaurantsViewModel`** is that we are enqueueing a **`Call`** object and we're waiting for the response through the **`Callback`** object, but we are never canceling that enqueued **`Call`**. We should cancel it when the host **`Activity`** or **`ViewModel`** is cleared to prevent memory leaks. Let's do that here:

1. Inside **`RestaurantsViewModel`**, define a class variable of type **`Call`** with the **`List<Restaurant>>`** type parameter. Call this variable **`restaurantsCall`** as we will use it to hold a reference to our enqueued **`Call`** object:

```
class RestaurantsViewModel(…): ViewModel() {
    private var restInterface:
RestaurantsApiService
    val state = […]
    private lateinit var restaurantsCall:
        Call<List<Restaurant>>
    init {…}
   […]
}
```

We've marked **`restaurantsCall`** as a **`lateinit`** variable to instantiate it later when we perform the network request.

2. Inside the **`getRestaurants()`** method of **`RestaurantsViewModel`**, assign the **`Call`** object that you obtained from the **`restInterface.getRestaurants()`** method call to the **`restaurantsCall`** member variable and call **`enqueue()`** on it:

```
fun getRestaurants() {
    restaurantsCall =
restInterface.getRestaurants()
    restaurantsCall.enqueue(object :
Callback<List<Restaurant>> {…})
}
```

3. Inside **`RestaurantsViewModel`**, override the **`onCleared()`** method and call the **`cancel()`** method of the **`restaurantCall`** object:

```
override fun onCleared() {
    super.onCleared()
    restaurantsCall.cancel()
}
```

The **`onCleared()`** callback method is provided by the Jetpack **`ViewModel`** and is called just before **`ViewModel`** is destroyed as a consequence of the attached activity/fragment or composable being destroyed or removed from the composition.

This callback represents the perfect opportunity for us to cancel any ongoing work – or in our case, to cancel the pending **`Call`** object that's enqueued in the **`restaurantCall`** object. This way, we prevent leaking memory and therefore fix the first issue in our code.

Now, it's time to focus on the second issue, where the **`RestaurantsScreen()`** composable calls the **`viewModel.getRestaurants()`** method without any special considerations.

Triggering network requests from a controlled environment

The `viewModel.getRestaurants()` method is called because we want to apply a **side effect** in our UI. A side effect is a change that's made to the state of the application that usually happens outside the scope of a composable function. In our case, the side effect is that we need to start loading the restaurants for the first time when the user enters the screen.

As a rule of thumb, composables should be side-effect free, but in our application, we need to know when to trigger the network request, and what better place than the moment when our UI is initially composed?

The problem with the existing approach of simply calling a method on `ViewModel` from the composable layer is that the Compose UI can be recomposed many times on the screen. For example, when an animation is rendered, the Compose UI is recomposed many times to execute the animation's keyframes. On every recomposition of the UI, our composable calls the `getRestaurants()` method on `RestaurantsViewModel`, which, in turn, executes network requests to obtain the restaurants from the server, which could result in multiple and redundant requests.

To prevent this issue from happening, Compose has the right tool for us to handle side-effects efficiently: the **Effects** API.

An *effect* is a composable function that, instead of emitting UI elements, causes side effects that run when a composition process completes. Such composables are based on the Kotlin Coroutine API, which allows you to run async work in their bodies. However, we will disregard coroutines for now as we will cover them in *Chapter 4*, *Handling Async Operations with Coroutines*.

In Compose, there are many types of effect composables that we can use but we will not go too deep into that. In our case, though, a suitable effect could be the `LaunchedEffect` composable since it allows us to run a task only once when it first enters composition.

The signature of **`LaunchedEffect`** is simple – it contains a **`key1`** parameter and a **`block`** parameter where we can execute our code. For now, we should ignore the Coroutine terminology and just think of the **`block`** function parameter as a block of code that can be executed asynchronously:

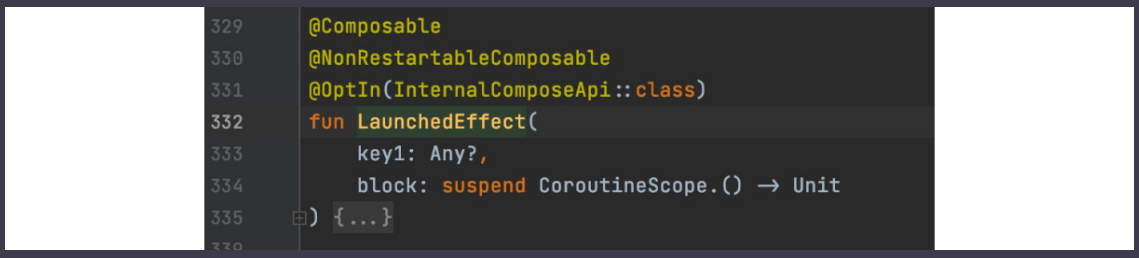

Figure 3.13 – The signature of the LaunchedEffect composable

When **`LaunchedEffect`** enters the composition process, it runs the **`block`** parameter function, which is passed as an argument. The execution of the block will be canceled if **`LaunchedEffect`** leaves the composition. If **`LaunchedEffect`** is recomposed with different keys that have been passed to the **`key1`** parameter, the existing execution of the block of code will be canceled and a new iteration of execution will be launched.

Now that we know how **`LaunchedEffect`** works, we can agree that it seems a viable solution for our issue, at least for now: we want to make sure that the call to **`ViewModel`** is only executed once on the initial composition, so **`LaunchedEffect`** seems to suffice our needs.

Let's add a **`LaunchedEffect`** to prevent our UI from asking for restaurants from **`ViewModel`** repeatedly on every recomposition:

1. Inside the **`RestaurantsScreen`** composable, wrap the **`viewModel.getRestaurants()`** call in a **`LaunchedEffect`** composable:

```
@Composable
fun RestaurantsScreen() {
    val viewModel: RestaurantsViewModel =
viewModel()
    LaunchedEffect(key1 = "request_restaurants") {
        viewModel.getRestaurants()
    }
```

```
        LazyColumn(…) { … }
    }
```

To implement the **`LaunchedEffect`** composable, we did the following:

- We passed a **`String`** hardcoded value of **`"request_restaurants"`** to the **`key1`** parameter. We passed a hardcoded value to the **`key1`** argument because we want the block of code passed inside the **`LaunchedEffect`** composable to not execute on every recomposition. We could have passed any constant to **`key1`**, yet what's important here is that the value shouldn't change over time.
- We passed our code that calls the **`getRestaurants()`** method on our **`ViewModel`** inside the **`block`** parameter of the effect. Since the **`block`** parameter is the last parameter of the **`LaunchedEffect`** composable and is a function, we used the trailing lambda syntax.

2. Run the application. Now, the code inside **`LaunchedEffect`** should only be executed once.

Yet even with this addition, our code still has an issue. If you try rotating the emulator or device you're testing with, you will trigger a configuration change and another network will be executed. But we mentioned previously that **`LaunchedEffect`** will only execute the **`viewModel.getRestaurants()`** call once, so why is this happening?

`LaunchedEffect` works fine – the issue lies in the activity being destroyed on configuration change. If the activity is destroyed, the UI will be composed again from scratch, and for all it knows, **`LaunchedEffect`** will run the code inside the **`block`** parameter for the first time.

Can you think of a better alternative to get around the issue of the activity being destroyed due to configuration changes?

An alternative would be to use the **`ViewModel`** component because it survives configuration changes. If we trigger the request only once in

RestaurantsViewModel, we no longer care if a configuration change occurs – the request will not be executed again. Follow these steps:

1. Inside **RestaurantsViewModel**, locate the **init** block and inside it, call **getRestaurants()**:

```
init {
    val retrofit: Retrofit = Retrofit.Builder().
[…].build()
    restInterface = retrofit.create(
        RestaurantsApiService::class.java
    )
    getRestaurants()
}
```

The **init** block is called once when an instance of **ViewModel** is created, so placing our network request here is a safer bet than at the UI level in any composable. Make sure you've placed the **getRestaurants()** call after the instantiation of the **restInterface** variable since the **getRestaurants()** method depends on that variable being ready to work.

2. Still inside **RestaurantsViewModel**, navigate to the **getRestaurants()** method and mark it as **private**:

```
private fun getRestaurants() {
    …
}
```

We no longer need to expose this method publicly to the UI since it's now only called inside **ViewModel**.

3. Inside the **RestaurantsScreen** composable, remove the **LaunchedEffect** composable function with all the code inside it since we no longer need it.
4. Run the application. The network request should not be executed again when a configuration change is made since the

`RestaurantsViewModel` instance is preserved and the code inside its `init` block is not executed again.

We've taken quite a few steps to make sure that our application handles network requests correctly, and this was a great first step toward creating a modern application.

Summary

In this chapter, we learned how mobile apps communicate with remote web APIs using HTTP connections and REST APIs. Then, we created a database for our Restaurants application with the help of Firebase and populated it with content.

After that, we explored what Retrofit is and how it abstracts the complexity associated with handling network requests and responses within HTTP connections between apps and web APIs.

Then, we executed a network request with Retrofit in our Restaurants application and learned how the JSON content that is sent by the server can be parsed or deserialized by our Retrofit networking client. We also learned how to correctly wait for network responses and how to notify the application when responses arrive.

Finally, we solved some common issues that occur when our applications communicate with web APIs asynchronously to retrieve data, especially in the context of Compose.

In the next chapter, we'll explore a very efficient tool in Android for async work that comes bundled with Kotlin: coroutines!

Further reading

With the help of custom annotations inside the Retrofit interface, this library hides most of the complexity associated with handling network requests. We've seen that with simple **GET** requests in our **RestaurantsApiService** interface when we annotated our request with the **@GET** annotation:

```
interface RestaurantsApiService {
    @GET("restaurants.json")
    fun getRestaurants(): Call<List<Restaurant>>
}
```

Yet apart from plain **GET** operations, such Retrofit interfaces can also handle other request types, such as **PUT, POST**, and **DELETE**.

For example, if you need to define a request that passes some data to the server that is likely to be stored, you can use a **POST** request by adding the **@POST** annotation to your desired method:

```
@POST("user/edit")
fun updateUser(@Field("first_name") firstName:
String):
    Call<User>
```

To understand how to use Retrofit for such cases, or more advanced ones, check out the official documentation: https://square.github.io/retrofit/.