Chapter 2: Deep Diving into Data Sources

In this chapter, we will study some of the popular libraries and frameworks used for retrieving and managing data on Android and how to do this without blocking the main thread of an application. We will start by going over how multithreading should be handled in an Android application and the available technologies we now have to easily handle this. We will then move on to implement loading data from the internet using libraries such as Retrofit and OkHttp, after which we will look at how we can persist data on a device using libraries such as Room and DataStore.

In this chapter, we will cover the following main topics:

- Understanding Kotlin coroutines and flows
- Using OkHttp and Retrofit for networking
- Using the Room library for data persistence
- Understanding and using the DataStore library

By the end of this chapter, you will have become familiar with how we can load, manage, and persist data in an Android application.

Technical requirements

This chapter has the following hardware and software requirements:

- Android Studio Arctic Fox 2020.3.1 Patch 3

The code files for this chapter can be found here:

https://github.com/PacktPublishing/Clean-Android-Architecture/tree/main/Chapter2 ↗️

Check out the following video to see the Code in Action:

https://bit.ly/38uecPi ↗️

Understanding Kotlin coroutines and Flows

In this section, we will look at how threading works in the Android ecosystem and what applications must do to ensure that long-running operations do not block the user from using an application. We will then look at what available options we have available to execute operations in the background, with a focus on coroutines. Finally, we will look over Kotlin flows, which we can use to handle asynchronous work using a reactive and functional approach.

Android applications normally run in a single process on a user's device. When the operating system starts the application's process, it will allocate memory resources for the process to be executed. This process, when started, will have one thread of execution running within. This thread is referred to as the "main thread" or "**user interface** (**UI**) thread". In Android, this concept is very important because it is the thread that deals with user interaction. This imposes certain limitations for developers when dealing with it, as outlined here:

- The main thread must not be blocked by long-running or **input/output** (**I/O**) operations.
- All updates to the UI must be done on the main thread.

The idea is that the user should still be able to interact with an application as much as possible even if the application is doing some work. Every time we want to load and save data from or to the internet, local storage, content providers, and so on, we should use another thread or use multiple threads. The way the device's processor deals with multiple threads is by assigning a core for each thread. When there are more threads than cores, it will jump back and forth between every single instruction from each thread. Having too many threads being executed simultaneously will end up creating a bad **user experience** (**UX**) because the processor will now need to jump between the main thread and the rest of the threads being executed at the same time, so we will need to be mindful of how many threads are being executed concurrently.

In Java, a thread can be created using the `Thread` class; however, creating a new thread for every asynchronous operation is a very resource-expensive operation. Java also offers the concept of `ThreadPool` or `Executor`. These typically manage fixed a collection of threads that will be reused for different operations. Because of the Android restriction regarding updating the UI on the main thread, classes such as `Handler` and `Looper` were introduced, whereby you can submit the result of an operation performed on a background thread back on the main thread. An example of this is provided here:

```
class MyClass {
    fun asyncSum(a: Int, b: Int, callback: (Int) ->
Unit) {
        val handler = Handler(Looper.getMainLooper())
        Thread(Runnable {
            val result = a + b
            handler.post(Runnable {
                callback(result)
            })
        }).start()
    }
}
```

In the preceding code snippet, the sum of two numbers will be performed on a new thread, and the result will then be posted back using the `Handler` object that is connected to the main `Looper` object, which itself will loop the main thread.

The repeated usage of `Handler` and `Looper` gave birth to `AsyncTask`, which offers the possibility of moving the necessary operations on a background thread and receiving the result on the main thread. **AsyncTask** worked with the same principle as the preceding example, only instead of creating a new thread for every new operation, it would by default use the same thread (although this later became configurable), which means that if two **AsyncTask** instances were executed at the same time, one would wait after the other. An example of the same sum operations might look like this:

```
    fun asyncSum(a: Int, b: Int, callback: (Int) ->
Unit) {
        object : AsyncTask<Nothing, Nothing, Int>() {
            override fun doInBackground(vararg
params:
                Nothing?): Int {
                return a+b
            }
            override fun onPostExecute(result: Int) {
                super.onPostExecute(result)
                callback(result)
            }
        }.execute()
    }
```

In the preceding example, the sum is done in the **doInBackground** method, which is executed on a separate thread, and the **onPostExecute** method would be executed on the main thread.

Let's now imagine that we want to chain these sums and apply them multiple times, as follows:

```
    fun asyncComplicatedSum(a: Int, b: Int, c: Int) {
        asyncSum(a, b) { tempSum ->
            asyncSum(tempSum, c) { finalSum ->
                Log.d(this.javaClass.name, "Final sum
                    $finalSum")
            }
            }
        }
```

In the preceding example, we try to sum two numbers and add the result to number **c**. As you can see, we need to use the callback and wait for **a** and **b** to finish and then apply the same function to the result of **a+b** and the number **c**.

Let's imagine what an application might look like when having to deal with loading data from multiple data sources, merging them together, handling errors, and stopping the asynchronous execution if the user leaves the current activity or fragment. The RxJava library tries to tackle all these problems through an event-driven approach. It introduces the concepts of streams and flows of data that can be observed, transformed, merged with other data streams, and executed on different threads. The sum of two numbers in RxJava might look something like this:

```
fun asyncSum(a: Int, b: Int): Single<Int> {
        return Single.create<Int> {
            it.onSuccess(a + b)
        }.subscribeOn(Schedulers.io())
            .observeOn(AndroidSchedulers.mainThread()
)
    }
```

In the preceding example, we create a **Single** instance, which is a stream that emits only one value (for emitting multiple values, we have the **Flowable** and **Observable** options). The value emitted is the sum of the two numbers. The usage of **subscribeOn** is for executing the upstream (the sum) on an I/O thread managed by RxJava internally, and the usage of **observeOn** is to have everything downstream (all the commands that will follow) to get the result on the main thread.

If we want to chain multiple sums, then we would have something like this:

```
fun asyncComplicatedSum(a: Int, b: Int, c: Int) {
```

```
            val disposable = asyncSum(a, b)
                .flatMap {
                    asyncSum(it, c)
                }
                .subscribe ({
                    Log.d(this.javaClass.name, "Final sum
$it")
                },{
                    Log.d(this.javaClass.name, "Something
went
                        wrong")
                })
    }
```

In the preceding example, the sum of **`a`** and **`b`** is executed, then through the **`flatMap`** operator, we add **`c`** to that result. The usage of **`subscribe`** method is for triggering sums and listening for the results. This is because the **`Single`** instance used is a cold observable; it will only be executed only when **`subscribe`** is called. There is also the concept of hot observables, which will emit whether there are subscribers or not. The result of the **`subscribe`** operator will return a **`Disposable`** instance that offers a **`dispose`** method that can be called when we want to stop listening for data from the stream. This is useful in situations where our activities and fragments are destroyed, and we don't want to update our UI to avoid context leaks.

Kotlin coroutines

So far, we have analyzed technologies that revolve around the Java and Android frameworks. With the adoption of Kotlin, other technologies have emerged that deal with multithreading and are Kotlin-specific. One of these is the concept of coroutines. Coroutines simplify the way we write asynchronous code. Instead of dealing with callbacks, coroutines introduce the concept of scopes where we can specify which threads our blocks of code will execute in. The scopes can also connect to lifecycle-aware components that help us unsubscribe from the results of asynchronous work when our lifecycle-aware components terminate. Let's look at the following example of coroutines for the same sum:

```
     suspend fun asyncSum(a: Int, b: Int): Int {
          return withContext(Dispatchers.IO) {
              a + b
          }
      }
```

In the preceding example, the **`withContext`** method will execute the block of code inside it in the threads managed by the I/O dispatcher. The number of threads associated with this dispatcher is managed internally by the Kotlin framework and is associated with the number of cores the processor of the device has. This often means that we don't have to worry about the performance of our applications when multiple asynchronous operations are executed concurrently. Another interesting thing to note in the example is the usage of the **`suspend`** keyword. This is to alert the caller of this method that it will be executed using coroutines on a separate thread.

Now, let's see what things will look like when we want to invoke this method. Have a look at the following code snippet:

```
  class MyClass : CoroutineScope {
      override val coroutineContext: CoroutineContext
          get() = Dispatchers.Main + job
      private lateinit var job: Job
      fun asyncComplicatedSum(a: Int, b: Int, c: Int) {
          launch {
              try {
                  val tempSum = asyncSum(a, b)
                  val finalSum = asyncSum(tempSum, c)
                  Log.d(this.javaClass.name, "Final sum
                      $finalSum")
              } catch (e: Exception) {
                  Log.d(this.javaClass.name, "Something
  went
                      wrong")
              }
          }
      }
      fun create() {
          job = Job()
      }
      fun destroy() {
          job.cancel()
      }
  }
```

In **`asyncComplicatedSum`**, we use the **`launch`** method. This method is associated with the **`CoroutineContext`** object defined in this class. The context is defined using the **`Main`** dispatcher combined with the **`Job`** object that will be associated with the lifecycle of this object. If the **`destroy`** method is called while we are waiting for the result of the sum, then the execution of the sum will stop and we will stop getting the result of the sum. The code will execute each of the sums on the I/O thread and then execute log statements on the main thread if the job is still alive.

In Android, we already have a few **`CoroutineScope`** objects already defined and associated with our lifecycle-aware classes. One that will be relevant to us is the one defined for **`ViewModels`**. This can be found in the **`org.jetbrains.kotlinx:kotlinx-coroutines-android`** library and will look something like this:

```
class MyViewModel: ViewModel() {
    init {
        viewModelScope.launch {  }
    }
}
```

`viewModelScope` is a Kotlin extension created for **`ViewModel`** instances that will execute if the **`ViewModel`** instance is alive. If **`onCleared`** is called on the

`ViewModel` instance, then it will stop listening to the remaining code to be executed in the `launch` block.

In this section, we've analyzed how Kotlin coroutines work and how we can use them to handle asynchronous operations in an Android application. In the next section, we will create an Android application that will use Kotlin coroutines for a simple asynchronous operation.

Exercise 02.01 – Using Kotlin coroutines

Create an application that will display two input fields, one text field, and a button. The input fields will be limited to numbers only, and when the user presses the button, then the text field will display the sum of the two numbers after 5 seconds. The sum and waiting will be implemented using coroutines.

To complete the exercise, you will need to build the following:

- A class that will perform the addition of the two numbers
- A `ViewModel` class that will invoke the addition
- The UI using Compose that will use the following function:

```
@Composable
fun Calculator(
    a: String,
    onAChanged: (String) -> Unit,
    b: String,
    onBChanged: (String) -> Unit,
    result: String,
    onButtonClick: () -> Unit
) {
    Column(modifier = Modifier.padding(16.dp)) {
        OutlinedTextField(
            value = a,
            onValueChange = onAChanged,
            keyboardOptions = KeyboardOptions
```

```
                (keyboardType =
KeyboardType.Number),
            label = { Text("a") }
        )
        OutlinedTextField(
            value = b,
            onValueChange = onBChanged,
            keyboardOptions = KeyboardOptions
                (keyboardType =
KeyboardType.Number),
            label = { Text("b") }
        )
        Text(text = result)
        Button(onClick = onButtonClick) {
            Text(text = "Calculate")
        }
    }
}
```

Follow these steps to complete the exercise:

1. Create a new project in Android Studio using an **Empty Compose Activity**.
2. At the top level of the **build.gradle** file, define the Compose library version as follows:

```
buildscript {
    ext {
        compose_version = '1.0.5'
    }
    …
}
```

3. In the **app/build.gradle** file, we need to add the following dependencies:

```
dependencies {
    implementation 'androidx.core:core-ktx:1.7.0'
    implementation
'androidx.appcompat:appcompat:1.4.0'
    implementation
'com.google.android.material:material:1.4.0'
    implementation
"androidx.compose.ui:ui:$compose_version"
    implementation
"androidx.compose.material:material:$compose_versio
n"
    implementation "androidx.compose.ui:ui-tooling-
preview:$compose_version"
    implementation 'androidx.lifecycle:lifecycle-
runtime-ktx:2.4.0'
    implementation 'androidx.activity:activity-
compose:1.4.0'
    implementation 'org.jetbrains.kotlinx:kotlinx-
coroutines-android:1.5.0'
    implementation "androidx.lifecycle:lifecycle-
viewmodel-ktx:2.4.0"
    implementation "androidx.lifecycle:lifecycle-
viewmodel-compose:2.4.0"
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation
'androidx.test.ext:junit:1.1.3'
    androidTestImplementation
'androidx.test.espresso:espresso-core:3.4.0'
    androidTestImplementation
"androidx.compose.ui:ui-test-
junit4:$compose_version"
    testImplementation
"org.jetbrains.kotlinx:kotlinx-coroutines-
test:1.5.0"
    debugImplementation "androidx.compose.ui:ui-
tooling:$compose_version"
}
```

4. Start by creating a **NumberAdder** class and define an **add** operation and a delay, as follows:

```
private const val DELAY = 5000
class NumberAdder(
    private val dispatcher: CoroutineDispatcher =
        Dispatchers.IO,
    private val delay: Int = DELAY
) {
    suspend fun add(a: Int, b: Int): Int {
        return withContext(dispatcher) {
            delay(delay.toLong())
            a + b
        }
    }
}
```

In this class, we will add our 5-second delay before performing the sum of the two numbers. This is to highlight the asynchronous operation more. **CoroutineDispatcher** and the amount we want to delay by will be injected through the constructor. This is because we want to unit-test this class.

5. Next, we will need to unit-test this class. Before we write the test, create a test rule so that we can reuse it for coroutines, as follows:

```
class DispatcherTestRule : TestRule {
    @ExperimentalCoroutinesApi
    val testDispatcher = TestCoroutineDispatcher()
    @ExperimentalCoroutinesApi
    override fun apply(base: Statement?,
description:
        Description?): Statement {
        try {
            Dispatchers.setMain(testDispatcher)
            base?.evaluate()
        } catch (e: Exception) {
        } finally {
            Dispatchers.resetMain()
            testDispatcher.cleanupTestCoroutines()
        }
        return base!!
    }
}
```

In this class, we create a **TestCoroutineDispatcher** instance that will later be injected into the unit test so that the test can execute the sum in a synchronous way. **@ExperimentalCoroutinesApi** suggests that the usage of **TestCoroutineDispatcher** is still in an experimental state and will be moved to a stable version in the future.

6. Now, write the unit test for the class, in the form of **NumberAdderTest**, as follows:

```
    class NumberAdderTest {
        @get:Rule
        val dispatcherTestRule = DispatcherTestRule()
        @ExperimentalCoroutinesApi
        @Test
        fun testAdd() = runBlockingTest {
            val adder = NumberAdder(dispatcherTestRule.
                testDispatcher, 0)
            assertEquals(5, adder.add(1, 4))
        }
    }
```

Here, we inject the **testDispatcher** object we created in **DispatcherTestRule** into **NumberAdder**, and we then invoke the **add** function. The entire test is executed in a special **CoroutineScope** block called **runBlockingTest**, that will ensure all the coroutines launched must complete.

7. Next, go ahead and create a **ViewModel** class, like this:

```
class MainViewModel(private val adder: NumberAdder
= NumberAdder()) : ViewModel() {
    var resultState by mutableStateOf("0")
        private set
    fun add(a: String, b: String) {
        viewModelScope.launch {
            val result = adder.add(a.toInt(),
                b.toInt())
            resultState = result.toString()
        }
    }
}
```

Here, we use a Compose state that will retain the result of the addition, and a method that will trigger the addition into **`viewModelScope`**.

8. After the **`ViewModel`** class has been created, go ahead and create an activity class, as follows:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Exercise201Theme {
                Surface {
                    Screen()
                }
            }
        }
    }
}
```

Here, we initialize our activity with the content. **`Exercise201Theme`** should be replaced with the theme generated by Android Studio when the project was created. Typically, this should be in a **`Theme`** file and should be a **`@Composable`** function that has the application name followed by the **`Theme`** suffix. If that is not available, you can use **`MaterialTheme`** instead for the purpose of the exercise.

9. Next, create a **`Screen`** function, as follows:

```
@Composable
fun Screen(viewModel: MainViewModel = viewModel())
{
    var a by remember { mutableStateOf("") }
    var b by remember { mutableStateOf("") }
    Calculator(
        a = a,
        onAChanged = {
            a = it
        },
        b = b,
        onBChanged = {
            b = it
        },
        result = viewModel.resultState,
        onButtonClick = {
            viewModel.add(a, b)
        })
}
```

In this method, we define variables for our text fields, then we pass the result of the addition of the numbers from the ViewModel, and finally, we invoke the ViewModel to perform the addition.

10. And finally, add the **`Calculator`** function from the exercise definition to the **`MainActivity`** file.

If we run the preceding example, we should see our UI elements, and after inserting the numbers and clicking the button, we will get our result. One thing to notice is that the user will be able to interact with the UI while the **add** method is executed, and clicking multiple times for different numbers will get the results 5 seconds after each button press.

Using coroutines can improve the quality of an Android application, especially when combined with Android extensions for the **ViewModel** class and lifecycle-aware components. Coroutines simplify the code we write for asynchronous operations, and the addition of the **suspend** keyword can enforce more rigor when dealing with these operations.

Kotlin Flows

Coroutines offer a good solution for dealing with asynchronous operations; however, they do not offer a good ability to handle multiple streams of data in the same way RxJava does. Flows represent an extension to coroutines, which is meant to solve this problem. When dealing with flows, there are three entities to consider, as outlined here:

- **Producer**: This entity is responsible for emitting the data.
- **Intermediary**: This entity deals with the transformation or manipulation of the data.
- **Consumer**: This entity consumes the data in the stream.

Let's look at the following example of adding two numbers and how it might look like using Kotlin flows:

```
fun asyncSum(a: Int, b: Int): Flow<Int> {
        return flow {
            this.emit(a + b)
        }.flowOn(Dispatchers.IO)
    }
```

Here, we create a **Flow** object that will emit the result of **a + b** on a stream. The **flowOn** method will move the execution of the upstream on an I/O thread. Here, we note the similarity to RxJava in the concept of how **Flows** work, but we also notice that it's an extension of coroutines because of the use of **`Dispatchers`**. Let's now look at how flows look on the consumer side, as follows:

```
class MyClass : CoroutineScope {
    override val coroutineContext: CoroutineContext
        get() = Dispatchers.Main + job
    private lateinit var job: Job
    @FlowPreview
    fun asyncComplicatedSum(a: Int, b: Int, c: Int) {
        launch {
            asyncSum(a, b)
                .flatMapConcat {
                    asyncSum(it, c)
                }
                .catch {
                    Log.d(this.javaClass.name,
"Something
                         went wrong")
                }
                .collect {
                    Log.d(this.javaClass.name, "Final
sum
                        $it")
                }
        }
    }
}
```

Here, we also notice similarities to RxJava—that is, when we try to manipulate the stream to perform the addition to number **`c`** and when it comes to error handling due to the **`catch`** method. The **`collect`** method, however, is closer to coroutines, and it requires a **`CoroutineScope`** to be used or to declare the calling method as a suspend one.

Flows offer a couple of specialized classes for particular use cases: **`StateFlow`** and **`SharedFlow`**. The **`StateFlow`** class is useful because it will offer subscribers the last value stored when they subscribe, like how **LiveData** works. Flows can also be cold and hot, and **SharedFlow** is a specialized implementation of a hot flow. **SharedFlow** will emit items if it is kept in memory if there are any consumers or not. When a consumer subscribes to **SharedFlow**, it will also emit the last value stored to the consumer, as with **StateFlow**.

In this section, we have looked at Kotlin flows and the benefits they provide when it comes to handling asynchronous operations. Next, we will look at how we can use Kotlin flows in an Android application through a simple exercise.

Exercise 02.02 – Using Kotlin Flows

Modify the application from *Exercise 02.01* so that the addition of the two numbers will return a Flow instead of a suspended function.

To complete the exercise, you will need to do the following:

- Rewrite the **add** function in **NumberAdder** to return a Flow.
- Change how the **ViewModel** invokes the **add** function.

Follow these steps to complete the exercise:

1. Modify the **add** function in **NumberAdder** to return a Flow, as follows:

```
private const val DELAY = 5000
class NumberAdder(
    private val dispatcher: CoroutineDispatcher =
        Dispatchers.IO,
    private val delay: Int = DELAY
) {
    suspend fun add(a: Int, b: Int): Flow<Int> {
        return flow {
            emit(a + b)
        }.onEach {
            delay(delay.toLong())
        }.flowOn(dispatcher)
    }
}
```

Here, we create a new Flow where we emit the sum of **a** and **b**, after which we put a delay on each item emitted in the stream, and finally, we specify the **CoroutineDispatcher** instance we wish to execute the sum on.

2. Next, let's modify the unit test for the sum, as follows:

```
class NumberAdderTest {
    @get:Rule
    val dispatcherTestRule = DispatcherTestRule()
    @ExperimentalCoroutinesApi
    @Test
    fun testAdd() = runBlockingTest {
        val adder = NumberAdder
            (dispatcherTestRule.testDispatcher, 0)
        val result = adder.add(1, 4).first()
        assertEquals(5, result)
    }
}
```

Because the **add** method returns a **Flow** object, we must now find the first item emitted in the flow and assert the value of that item against our expected result.

3. Modify the **MainViewModel** class to consume the **add** operation, as follows:

```
class MainViewModel(private val adder: NumberAdder
= NumberAdder()) : ViewModel() {
    var resultState by mutableStateOf("0")
        private set
    fun add(a: String, b: String) {
        viewModelScope.launch {
            adder.add(a.toInt(), b.toInt())
                .collect {
                    resultState = it.toString()
                }
        }
    }
}
```

Here, the **`add`** method will still use the same **`CoroutineScope`** instance to launch the **`add`** method, which will now use the **`collect`** method to get the result of the sum.

If we launch the application after following the steps from the exercise, the behavior will be the same as for *Exercise 02.01*, and we can see how Kotlin flows extend the functionality of coroutines by introducing concepts from RxJava to simplify how we can handle multiple streams of data.

In this section, we've seen how handling asynchronous operations has evolved over time and how much our applications benefit from concepts such as coroutines and flows that provide management for background threads, simplify how we execute asynchronous operations, manage multiple streams of data, and can connect to the lifecycle of Android components. In the following section, we will look at tools we can use to fetch data from the network and how they can be integrated with Kotlin coroutines and flows.

Using OkHttp and Retrofit for networking

In this section, we will look at how we can use the Retrofit library to perform networking operations and the benefits it provides.

Many Android applications require the internet to access data stored on various servers. Often, this is done through the **HyperText Transfer Protocol** (**HTTP**) protocol in which data is exchanged between the applications and the servers. The data is often represented in **JavaScript Object Notation** (**JSON**) format. In the past, these types of exchanges were implemented either with `HttpURLConnection` or Apache HttpClient. Working with either of these components meant that developers would need to manually handle the conversion from **plain old Java objects** (**POJOs**) to JSON, handle various network configurations, and deal with backward compatibility.

The OkHttp library will address some of these issues through an `OkHttpClient` class that will handle various network configurations and that provides other features such as caching. The Retrofit library, which can be placed on top of the OkHttp library, is meant to ensure type safety when dealing with various data formats. It's very configurable and allows the possibility to plug in various converter libraries for POJO-to-JSON conversion or **Extensible Markup Language** (**XML**) or other types of formats.

In order to add Retrofit and OkHttp to the project, we will add the following dependencies to the `build.gradle` file:

```
dependencies {
    …
    implementation
"com.squareup.okhttp3:okhttp:4.9.0"
    implementation
"com.squareup.retrofit2:retrofit:2.9.0"
    …
}
```

Next, we will need to determine which converters we will need to use for the data. Because JSON is a common format, we will use a JSON converter and the Moshi library to do so, so we will need to add dependencies to these two libraries, as follows:

```
dependencies {
    …
    implementation
"com.squareup.okhttp3:okhttp:4.9.0"
    implementation
"com.squareup.retrofit2:retrofit:2.9.0"
    implementation "com.squareup.retrofit2:converter-
moshi:2.9.0"
    implementation "com.squareup.moshi:moshi:1.13.0"
    …
}
```

Here, the Moshi library will be responsible for converting POJOs into JSON, and the converter library will plug into the Retrofit library and trigger this conversion when data is exchanged between the Android application and the server.

Let's assume we will need to fetch data from a server in a JSON format. We can use the https://jsonplaceholder.typicode.com/ ↗️ service as an example. If we want to fetch a list of users, we can use the https://jsonplaceholder.typicode.com/users ↗️ **Uniform Resource Locator** (**URL**). A user's JSON representation looks like this:

```
{
"id": 1,
    "name": "Leanne Graham",
    "username": "Bret",
    "email": "Sincere@april.biz",
    "address": {
      "street": "Kulas Light",
      "suite": "Apt. 556",
      "city": "Gwenborough",
      "zipcode": "92998-3874",
      "geo": {
        "lat": "-37.3159",
        "lng": "81.1496"
      }
    },
    "phone": "1-770-736-8031 x56442",
    "website": "hildegard.org",
    "company": {
      "name": "Romaguera-Crona",
      "catchPhrase": "Multi-layered client-server
neural-
          net",
      "bs": "harness real-time e-markets"
    }
```

We can see in the JSON representation that the user has an **`id`**, a **`username`**, an **`email`** value, and so on. In Kotlin, we can create a representation of this, and we can exclude properties that the application doesn't need, such as **`email`**, **`address`**, **`phone`**, **`website`**, and **`company`**, as follows:

```
    data class User(
        @Json(name = "id") val id: Long,
        @Json(name = "name") val name: String,
        @Json(name = "username") val username: String
    )
```

Here, we are using Moshi to map the property from a JSON to a Kotlin type, and we only kept three of the fields present in the initial JSON. Now, let's look at how we can initialize our networking libraries. The code to accomplish this is shown in the following snippet:

```
  fun createOkHttpClient() =  OkHttpClient
        .Builder()
        .readTimeout(15, TimeUnit.SECONDS)
        .connectTimeout(15, TimeUnit.SECONDS)
        .build()
```

For OkHttp, we use a **`Builder`** method to create a new **`OkHttpClient`** instance, and we can provide certain configurations for it. We will now use the **`OkHttpClient`** instance created previously to create a **`Retrofit`** instance, as follows:

```
fun createRetrofit(
        okHttpClient: OkHttpClient
    ): Retrofit {
        return Retrofit.Builder()
            .baseUrl("https://jsonplaceholder.typicod
e.com/")
            .client(okHttpClient)
            .build()
    }
```

Here, we create a new **`Retrofit`** instance that will have the base URL set to https://jsonplaceholder.typicode.com/ ↗. Changing the base URL comes in handy during development. Many teams will have a development URL used internally to test the development and integration of features and will have a production URL where the actual user data is set. Now, we will need to connect the Moshi JSON serialization to the **`Retrofit`** instance, as follows:

```
Fun createConverterFactory(): MoshiConverterFactory =
MoshiConverterFactory.create()
```

Here, we create **`MoshiConverterFactory`**, which is a Retrofit converter designed to connect **`Retrofit`** to the JSON serialization done by Moshi. We will now need to change our **`Retrofit`** initialization to what follows:

```
fun createRetrofit(
        okHttpClient: OkHttpClient,
        gsonConverterFactory: MoshiConverterFactory
    ): Retrofit {
        return Retrofit.Builder()
.baseUrl("https://jsonplaceholder.typicode.com/")
            .client(okHttpClient)
            .addConverterFactory(gsonConverterFactory
)
            .build()
    }
```

Here, we add the **`MoshiConverterFactory`** converter to the Retrofit **`Builder`** method to allow the two components to work together. Finally, we can create a Retrofit interface that will have templates for the HTTP request, as follows:

```
interface UserService {
        @GET("/users")
        fun getUsers(): Call<List<User>>
        @GET("/users/{userId}")
        fun getUser(@Path("userId") userId: Int):
            Call<User>
        @POST("/users")
        fun createUser(@Body user: User): Call<User>
        @PUT("/users/{userId}")
        fun updateUser(@Path("userId") userId: Int,
@Body
            user: User): Call<User>
    }
```

This interface contains an example of various methods for getting, creating, updating, and deleting data on servers. Note that the return type of these methods is a **`Call`** object that offers the ability to execute HTTP requests synchronously or asynchronously. One of the things that makes Retrofit more appealing to developers is the fact that it can be integrated with other asynchronous libraries such as RxJava and coroutines. Translating the preceding example to coroutines will look something like this:

```
interface UserService {
        @GET("/users")
        suspend fun getUsers(): List<User>
        @GET("/users/{userId}")
        suspend fun getUser(@Path("userId") userId:
Int):
            User
        @POST("/users")
        suspend fun createUser(@Body user: User):
User
        @PUT("/users/{userId}")
        suspend fun updateUser(@Path("userId")
userId: Int,
            @Body user: User): User
    }
```

In the preceding example, we add the **suspend** keyword to each method and we remove the dependency to the **Call** class. This allows us to execute these methods using coroutines. To create an instance of this class, we need to do the following:

```
fun createUserService(retrofit: Retrofit) =
retrofit.create(UserService::class.java)
```

Here, we use the **Retrofit** instance created previously to create a new instance of **UserService**.

In this section, we have analyzed how we can use OkHttp and Retrofit to load data from the internet and the benefits these libraries provide, especially when combined with Kotlin coroutines and flows. In the next section, we will create an Android application that will use these libraries to fetch and display data on the UI.

Exercise 02.03 – Using OkHttp and Retrofit

Create an Android application that connects to https://jsonplaceholder.typicode.com/ ↗ and displays a list of users using OkHttp, Retrofit, and Moshi. For each user, we will display the name, username, and email.

To complete the exercise, you will need to do the following:

- Create a **User** data class that will map the JSON representation of the user.

- Create a **`UserService`** class that will have a method to retrieve a list of users.
- Create a **`ViewModel`** class that will use **`UserService`** to retrieve a list of users.
- Implement an **`Activity`** class that will display a list of users.

A UI list will be created using the following method:

```
@Composable
fun UserList(users: List<User>) {
    LazyColumn(modifier = Modifier.padding(16.dp)) {
        items(users) {
            Column(modifier =
Modifier.padding(16.dp)) {
                Text(text = it.name)
                Text(text = it.username)
                Text(text = it.email)
            }
        }
    }
}
```

Follow these steps to complete the exercise:

1. Create an Android application with an **Empty Compose Activity**.
2. At the top level of the **`build.gradle`** file, define the Compose library version, as follows:
   ```
   buildscript {
       ext {
           compose_version = '1.0.5'
       }
       …
   }
   ```
3. In the **`app/build.gradle`** file, add the following dependencies:
   ```
   dependencies {
       implementation 'androidx.core:core-ktx:1.7.0'
   ```

```
    implementation
'androidx.appcompat:appcompat:1.4.0'
    implementation
'com.google.android.material:material:1.4.0'
    implementation
"androidx.compose.ui:ui:$compose_version"
    implementation
"androidx.compose.material:material:$compose_versio
n"
    implementation "androidx.compose.ui:ui-tooling-
preview:$compose_version"
    implementation 'androidx.lifecycle:lifecycle-
runtime-ktx:2.4.0'
    implementation 'androidx.activity:activity-
compose:1.4.0'
    implementation 'org.jetbrains.kotlinx:kotlinx-
coroutines-android:1.5.0'
    implementation "androidx.lifecycle:lifecycle-
viewmodel-ktx:2.4.0"
    implementation "androidx.lifecycle:lifecycle-
viewmodel-compose:2.4.0"
    implementation
"com.squareup.okhttp3:okhttp:4.9.0"
    implementation
"com.squareup.retrofit2:retrofit:2.9.0"
    implementation
"com.squareup.retrofit2:converter-moshi:2.9.0"
    implementation
"com.squareup.moshi:moshi:1.13.0"
    implementation "com.squareup.moshi:moshi-
kotlin:1.13.0"
    testImplementation 'junit:junit:4.13.2'
    androidTestImplementation
'androidx.test.ext:junit:1.1.3'
```

```
    androidTestImplementation
'androidx.test.espresso:espresso-core:3.4.0'
    androidTestImplementation
"androidx.compose.ui:ui-test-
junit4:$compose_version"
    testImplementation
"org.jetbrains.kotlinx:kotlinx-coroutines-
test:1.5.0"
    debugImplementation "androidx.compose.ui:ui-
tooling:$compose_version"
}
```

4. Now, add a permission for internet access to the **`AndroidManifest.xml`** file, as follows:

```
<uses-permission
android:name="android.permission.INTERNET"/>
```

5. Now move on and create a class that will hold the user information, as follows:

```
@JsonClass(generateAdapter = true)
data class User(
    @Json(name = "id") val id: Long,
    @Json(name = "name") val name: String,
    @Json(name = "username") val username: String,
    @Json(name = "email") val email: String
)
```

Here, we will hold the **`id`** field, which is generally a relevant field for distinguishing between different users and fields that we are required to display.

6. Next, create a **`UserService`** class that will fetch the user data, as follows:

```
interface UserService {
    @GET("/users")
    suspend fun getUsers(): List<User>
}
```

Here, we will only have one method that will get a list of users from the **`/users`** path.

7. Now, we initialize the networking objects. Because we aren't using any **dependency injection** (**DI**) frameworks and we only need to create one instance of each, we will hold the objects in the **`MainApplication`** class, as follows:

```
class MyApplication : Application() {
    companion object {
        lateinit var userService: UserService
    }
    override fun onCreate() {
        super.onCreate()
        val okHttpClient = OkHttpClient
            .Builder()
            .readTimeout(15, TimeUnit.SECONDS)
            .connectTimeout(15, TimeUnit.SECONDS)
            .build()
        val moshi = Moshi.Builder().
            add(KotlinJsonAdapterFactory()).build()
        val retrofit = Retrofit.Builder()
            .baseUrl("https://jsonplaceholder.typic
ode.com/")
            .client(okHttpClient)
            .addConverterFactory(MoshiConverterFact
ory.create(moshi))
            .build()
        userService =
retrofit.create(UserService::class.java)
    }
}
```

Here, we are initializing our networking libraries and the **`UserService`** object. Currently, we are holding a static reference to this object, which is not a good idea in general. Normally, we would rely on DI frameworks to manage these networking dependencies.

8. In the **`AndroidManifest.xml`** file, add the following code:

```
      <application
            …
            android:name=".MyApplication"
            …>
```

Given that we are inheriting from the **`Application`** class, we will need to add this class to the manifest.

9. Next, go ahead and create a **`MainViewModel`** class, as follows:

```
    class MainViewModel(private val userService:
        UserService) : ViewModel() {
        var resultState by mutableStateOf
            <List<User>>(emptyList())
            private set
        init {
            viewModelScope.launch {
                    val users = userService.getUsers()
                    resultState = users
            }
        }
    }
    class MainViewModelFactory :
    ViewModelProvider.Factory {
        override fun <T : ViewModel> create(modelClass:
            Class<T>): T =
            MainViewModel(MyApplication.userService) as
    T
    }
```

The **`MainViewModel`** class will depend on the **`UserService`** class to get a list of **`Users`** and store them in a Compose state that will be used in the UI.

Here, we are also creating a **MainViewModelFactory** class that will be responsible for injecting the **UserService** class into the **MainViewModel** class.

10. Now, we move on and create a **MainActivity** class, as follows:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            Exercise0203Theme {
                Surface {
                    Screen()
                }
            }
        }
    }
}
```

Here, we initialize our activity with the content. The **Exercise203Theme** theme should be replaced with the theme generated by Android Studio when the project was created. Typically, this should be in a **Theme** file and should be a **@Composable** function that has the application name followed by the **Theme** suffix. If that is not available, you can use **MaterialTheme** instead for the purpose of the exercise.

11. Create a **Screen** method in which we will grab a list of users from the **MainViewModel** class and draw a list of items, as follows:

```
@Composable
fun Screen(viewModel: MainViewModel = viewModel
    (factory = MainViewModelFactory())) {
    UserList(users = viewModel.resultState)
}
```

12. And finally, add the **UserList** function from the exercise definition into the **MainActivity** file.

If we launch the application after following the steps from the exercise, we should be able to see a list of users being loaded if the device has internet access.

In this section, we have seen how we can typically retrieve data from the internet in an Android application. We have looked at libraries such as OkHttp and Retrofit and seen how straightforward it is to make HTTP calls in a type-safe way without converting JSON files to data classes manually. We have also observed the potential of these libraries due to their integration with asynchronous technologies such as RxJava and coroutines. In the following section, we will look at libraries used for persisting data and how we can integrate them with networking libraries as well as coroutines and flows.

Using the Room library for data persistence

In this section, we will discuss how to persist data in Android applications and how we can use the Room library to do this.

Android offers many ways for persisting data on an Android device, mostly involving files. Some of these files have a specialized approach to persisting data. One of these approaches is in the form of SQLite. SQLite is a special type of file in which structured data can be stored using **Structured Query Language** (**SQL**) queries, as with other types of databases such as MySQL and Oracle.

In the past, if developers wanted to persist data in SQLite, they were required to manually define tables, write queries, and transform objects containing this data into the appropriate formats for performing **create, read, update, and delete** (**CRUD**) operations. This type of work involved a load of boilerplate code that was susceptible to bugs. Room is the answer to that by providing an abstraction layer on top of the SQLite operations.

In order to add Room to an application, we will need to add the following libraries in **`build.gradle`**:

```
dependencies {
    …
    implementation "androidx.room:room-runtime:2.4.0"
    kapt "androidx.room:room-compiler:2.4.0"
    …
}
```

The reason for the **`kapt`** usage is that Room uses annotations that will generate the code required for the interaction with the SQLite layer. In order to use the **`kapt`** feature, we will need to add the plugin to the **`build.gradle`** file, as follows:

```
plugins {
    …
    id 'kotlin-kapt'
}
```

This will allow the build system to analyze annotations across the project that require code generation and generate the necessary classes based on the provided annotations.

The data we want to store is annotated with the **`@Entity`** annotation, as illustrated in the following code snippet:

```
@Entity(tableName = "user")class UserEntity(
    @PrimaryKey @ColumnInfo(name = "id") val id:
Long,
    @ColumnInfo(name = "name") val name: String,
    @ColumnInfo(name = "username") val username:
String
)
```

Here, we have defined a Room entity named **`UserEntity`** that will represent a table named **`user`** and has the **primary key** (**PK**) set to be the **iden-**

tifier (**ID**) of the user. The `@ColumnInfo` annotation is for the name the column will have in the database.

A typical set of CRUD operations might look like this:

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): List<UserEntity>
    @Query("SELECT * FROM user WHERE id IN
(:userIds)")
    fun loadAllByIds(userIds: IntArray):
List<UserEntity>
    @Insert
    fun insert(vararg users: User)
    @Update
    fun update(vararg users: User)
    @Delete
    fun delete(user: User)
}
```

Just as how we defined in Retrofit a service interface to communicate with the server, we also define a similar interface for Room that we annotate with `@Dao`, for **data access object** (**DAO**). In this example, we have defined a set of functions for getting all users stored in a table, finding users, inserting new users, updating a user, and deleting a user.

As with Retrofit, Room also provides integrations with coroutines, as illustrated in the following code snippet:

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    suspend fun getAll(): List<UserEntity>
    @Query("SELECT * FROM user WHERE id IN
(:userIds)")
```

```
    suspend fun loadAllByIds(userIds: IntArray):
        List<UserEntity>
    @Insert
    suspend fun insert(vararg users: User)
    @Update
    suspend fun update(vararg users: User)
    @Delete
    suspend fun delete(user: User)
}
```

In the preceding example, we add the **`suspend`** keyword, which makes the Room library easy to integrate and execute as part of a coroutine.

On top of coroutines, the Room library also can integrate with Kotlin flows. This is useful for queries that will emit events every time a particular table has changed. This integration will look something like this:

```
@Dao
interface UserDao {
    @Query("SELECT * FROM user")
    fun getAll(): Flow<List<UserEntity>>
    @Query("SELECT * FROM user WHERE id IN
(:userIds)")
    fun loadAllByIds(userIds: IntArray):
        Flow<List<UserEntity>>
}
```

In the preceding example, we have changed the **`@Query`** functions to return a **`Flow`** object. If a change occurs in the user table, then the queries will be re-triggered and a new list of users will be emitted.

We will now need to set up the database, as follows:

```
@Database(entities = [UserEntity::class], version =
1)
abstract class AppDatabase : RoomDatabase() {
    abstract fun userDao(): UserDao
}
```

In the preceding snippet, we define a new class that extends from the **`RoomDatabase`** class and use the **`@Database`** annotation to declare our entities and the current version. This version is used to keep track of migrations when the structure of the database changes in between new releases of our application.

To initialize the database, we will need to execute the following code:

```
val db = Room.databaseBuilder(
            applicationContext,
            AppDatabase::class.java, "name"
        ).build()
```

This will create our SQLite database and will return an instance of **`AppDatabase`** where we can access the DAO objects we have defined and invoke their methods to process the data.

In this section, we have looked at how we can persist data using Room and how it can be integrated with coroutines and flows. In the next section, we will create an Android application that will use Room to persist data and look at how it can be integrated with Retrofit and OkHttp.

Exercise 02.04 – Using Room to persist data

Integrate Room into *Exercise 02.03* so that when the users are loaded from Retrofit, they will be stored in the database and then displayed on the UI.

To complete the exercise, you will need to do the following:

1. Create a **`UserEntity`** class that will be a Room entity.
2. Create a **`UserDao`** class that will contain methods for inserting users and querying all the users as flows.
3. Create an **`AppDatabase`** class that will represent the application's database.

4. Modify the **MainViewModel** class to fetch users from the **UserService** class and then insert them into the **UserDao** class.
5. Modify the **MainActivity** class to use a list of **UserEntity** objects instead of **User** objects.

Follow these steps to complete the exercise:

1. Add the **kapt** plugin to the **app/build.gradle** file, as follows:
```
plugins {
    …
    id 'kotlin-kapt'
}
```
2. Add Room dependencies to **app/build.gradle**, as follows:
```
dependencies {
    …
    implementation "androidx.room:room-
runtime:2.4.0"
    implementation "androidx.room:room-ktx:2.4.0"
    kapt "androidx.room:room-compiler:2.4.0"
    …
}
```
3. Create a **UserEntity** class, as follows:
```
@Entity(tableName = "user")
class UserEntity(
    @PrimaryKey @ColumnInfo(name = "id") val id:
Long,
    @ColumnInfo(name = "name") val name: String,
    @ColumnInfo(name = "username") val username:
        String,
    @ColumnInfo(name = "email") val email: String
)
```

The **UserEntity** class has the same fields as the **User** class, and it contains the Room annotations for the table name and the names of each column.

4. Next, create a **`UserDao`** class, as follows:
   ```
   @Dao
   interface UserDao {
       @Query("SELECT * FROM user")
       fun getUsers(): Flow<List<UserEntity>>
       @Insert(onConflict =
   OnConflictStrategy.REPLACE)
       fun insertUsers(users: List<UserEntity>)
   }
   ```

Here, we are using flows to return a list of users, and we use the **`OnConflictStrategy.REPLACE`** option so that if the same user is inserted multiple times, then it will be replaced with the one that will be inserted. Other options include **`OnConflictStrategy.ABORT`**, which will drop the entire transaction if a conflict occurs, or **`OnConflictStrategy.IGNORE`**, which will skip inserting rows where a conflict occurs.

5. Now, go ahead and create an **`AppDatabase`** class, as follows:
   ```
   @Database(entities = [UserEntity::class], version =
   1)
   abstract class AppDatabase : RoomDatabase() {
       abstract fun userDao(): UserDao
   }
   ```

In **`AppDatabase`**, we provide the **`UserDao`** class to be accessed and we use the **`UserEntity`** class for the users' table.

6. Next, we will need to initialize the **`AppDatabase`** object, as follows:
   ```
   class MyApplication : Application() {
       companion object {
           …
           lateinit var userDao: UserDao
           …
       }
       override fun onCreate() {
   ```

```
            super.onCreate()
            …
            val db = Room.databaseBuilder(
                applicationContext,
                AppDatabase::class.java, "my-database"
            ).build()
            userDao = db.userDao()
            …
        }
    }
```

Here, we are having the same issues that we had for Retrofit, so we will follow the same approach and use the **`Application`** class. Just as with Retrofit, a DI framework will help us solve this problem.

7. Now, let's integrate Room into the **`MainViewModel`** class, as follows:

```
    class MainViewModel(
        private val userService: UserService,
        private val userDao: UserDao
    ) : ViewModel() {
        var resultState by
    mutableStateOf<List<UserEntity>>(emptyList())
            private set
        init {
            viewModelScope.launch {
                flow { emit(userService.getUsers()) }
                    .onEach {
                    val userEntities =
                        it.map { user -> UserEntity
                            (user.id, user.name,
                             user.username, user.email)
    }
                    userDao.insertUsers(userEntities)
                }.flatMapConcat { userDao.getUsers() }
                    .catch {
    emitAll(userDao.getUsers()) }
                    .flowOn(Dispatchers.IO)
                    .collect {
                        resultState = it
                    }
            }
        }
    }
    class MainViewModelFactory :
    ViewModelProvider.Factory {
        override fun <T : ViewModel> create(modelClass:
    Class<T>): T =
            MainViewModel(MyApplication.userService,
    MyApplication.userDao) as T
    }
```

The **MainViewModel** class now has a new dependency on the **UserDao** class. In the **init** block, we now create a flow in which we emit a list of users obtained from Retrofit that is then converted into **UserEntity** and inserted into the database. After this, we will query the **UserEntities** instances and return them in a stream that will be the result. If we have an error, we will return the current stored users.

8. Finally, update the type of users in the **MainActivity** class, as follows:

```
class MainActivity : ComponentActivity() {
…
@Composable
fun UserList(users: List<UserEntity>) {
…
    }
}
```

Here, we just change the dependency to now rely on the **UserEntity** class.

If we run the application after following the steps from the exercise, we will see the same output as for *Exercise 02.03*. However, if we close the application, turn on Airplane mode on the device, and reopen the app, we will still see the previously displayed information.

In this section, we have analyzed how we can persist structured data on a device and used the Room library to do so. We have also observed the interaction between Room and other libraries such as Retrofit and flows and how we can use flows to combine data streams from Room and Retrofit in a very straightforward way. In the next section, we will look at how we can persist simple data in key-value pairs.

Understanding and using the DataStore library

In this section, we will discuss how we can persist key-value pairs of data and how we can use the DataStore library for this. In Android, we have the possibility of persisting primitives and strings in key-value pairs. In the past, this was done through the **`SharedPreferences`** class, which was part of the Android framework. The keys and values would ultimately be saved inside an XML file on the device. Because this deals with I/O operations, it evolved over time to give the possibility to save data asynchronously and to keep an in-memory cache for quick access to data. There were, however, some inconsistencies with this, especially when the **`SharedPreferences`** object was initialized. DataStore is designed to address these issues because it's integrated with coroutines and flows.

To add DataStore to a project, we will need the following dependency:

```
dependencies {
    …
    implementation "androidx.datastore:datastore-
preferences:1.0.0"
    …
}
```

Using DataStore will look something like this:

```
private val KEY_TEXT =
stringPreferencesKey("key_text")
class AppDataStore(private val dataStore:
    DataStore<Preferences>) {
    val savedText: Flow<String> = dataStore.data
        .map { preferences ->
            preferences[KEY_TEXT].orEmpty()
        }
    suspend fun saveText(text: String) {
        dataStore.edit { preferences ->
            preferences[KEY_TEXT] = text
        }
    }
}
```

The **`KEY_TEXT`** field will represent a key that will be used to store some text. **`DataStore<Preferences>`** is responsible for obtaining and writing the data to **`SharedPreferences`**. The **`savedText`** field will monitor changes in the preferences and will emit a new value for each change in a **`Flow`** object. To write data in an asynchronous way, we will need to edit the current data store and set the value associated with the key.

To initialize the DataStore library, we will need to declare the following as a top-level declaration:

```
val Context.dataStore: DataStore<Preferences> by
preferencesDataStore(name = "my_preferences")
```

This will allow us to access the DataStore library in the rest of the application.

When we want to initialize **`AppDataStore`**, we can use the following code:

```
val appDataStore = AppDataStore(dataStore)
```

This allows us to wrap the **DataStore** class and avoid exposing the dependencies to other places in the application.

In this section, we have looked at how we can persist data in key-value pairs and how we can use the DataStore library to do this. In the next section, we will create an Android application that will use DataStore and integrate it with Kotlin flows and coroutines.

Exercise 02.05 – Using DataStore to persist data

Modify *Exercise 02.04* and introduce the DataStore library, which will persist the number of executed requests to get the user and display this number above the list of items.

To complete the exercise, you will need to do the following:

- Create a class named **AppDataStore** that will manage interaction with the DataStore library.
- Modify the **MainViewModel** class so that the **AppDataStore** dependency is injected and used to retrieve the current number of requests and increment the number of requests.
- Modify the **MainActivity** class to add a new **Text** object that will display the count of requests.

Follow these steps to complete the exercise:

1. Add the following dependency to the **app/build.gradle** file:
```
    dependencies {
        …
        implementation "androidx.datastore:datastore-
            preferences:1.0.0"
        …
    }
```
2. Create an **AppDataStore** class, as follows:

```
private val KEY_COUNT =
intPreferencesKey("key_count")
class AppDataStore(private val dataStore:
    DataStore<Preferences>) {
    val savedCount: Flow<Int> = dataStore.data
        .map { preferences ->
            preferences[KEY_COUNT] ?: 0
        }
    suspend fun incrementCount() {
        dataStore.edit { preferences ->
            val currentValue =
preferences[KEY_COUNT]
                ?: 0
            preferences[KEY_COUNT] = currentValue.
                inc()
        }
    }
}}
```

Here, **`KEY_COUNT`** represents the key used by the DataStore library to store the number of requests. The **`saveCount`** field will emit a new count value every time it changes, and **`incrementCount`** will be increment the current saved number by 1.

3. Now, set up the **`AppDataStore`** dependency, just like how we handled the Retrofit and Room dependencies. The code is illustrated in the following snippet:

```
val Context.dataStore: DataStore<Preferences> by
preferencesDataStore(name = "my_preferences")
class MyApplication : Application() {
    companion object {
        …
        lateinit var appDataStore: AppDataStore
    }
    override fun onCreate() {
        super.onCreate()
        …
        appDataStore = AppDataStore(dataStore)
    }
}
```

Here, we initialize the **`DataStore`** object and then inject it into the **`AppDataStore`** class.

4. Next, modify the **`MainViewModel`** class, as follows:

```
    class MainViewModel(
        private val userService: UserService,
        private val userDao: UserDao,
        private val appDataStore: AppDataStore
    ) : ViewModel() {
        var resultState by mutableStateOf(UiState())
            private set
        init {
            viewModelScope.launch {
                flow { emit(userService.getUsers()) }
                    .onEach {
                        val userEntities =
                            it.map { user -> UserEntity
                                (user.id, user.name,
    user.
                                 username, user.email)
    }
                        userDao.insertUsers(userEntitie
    s)
                        appDataStore.incrementCount()
                    }.flatMapConcat {
    userDao.getUsers() }
                    .catch {
    emitAll(userDao.getUsers()) }
                    .flatMapConcat { users ->
```

```
                            appDataStore.savedCount.map {
                                count -> UiState(users,
                                    count.toString()) }
                        }
                        .flowOn(Dispatchers.IO)
                        .collect {
                            resultState = it
                        }
                }
            }
        }
```

Here, we add a new dependency to **`AppDataStore`**, then we call **`increment-Count`** from **`AppDataStore`** after the users from Retrofit are inserted, and then we will insert **`savedCount`** from **`AppDataStore`** into the existing flow and create a new **`UiState`** object that contains a list of users and the count, which will be collected in the **`resultState`** object.

5. The **`UiState`** class will look something like this:

```
data class UiState(
    val userList: List<UserEntity> = listOf(),
    val count: String = ""
)
```

This class will hold information from both of our persistent data sources.

6. Next, change **`MainViewModelFactory`**, as follows:

```
class MainViewModelFactory :
ViewModelProvider.Factory {
    override fun <T : ViewModel> create(modelClass:
        Class<T>): T =
        MainViewModel(
            MyApplication.userService,
            MyApplication.userDao,
            MyApplication.appDataStore
            ) as T
    }
```

Here, we will inject a new dependency to **AppDataStore** into the **MainViewModel** class.

7. Finally, modify the **MainActivity** class, as follows:

```
    @Composable
    fun UserList(uiState: UiState) {
        LazyColumn(modifier = Modifier.padding(16.dp))
    {
            item(uiState.count) {
                Column(modifier =
    Modifier.padding(16.dp)) {
                    Text(text = uiState.count)
                }
            }
            items(uiState.userList) {
                Column(modifier =
    Modifier.padding(16.dp)) {
                    Text(text = it.name)
                    Text(text = it.username)
                    Text(text = it.email)
                }
            }
        }
    }
```

Here, we replaced the list of **UserEntity** with the **UiState** dependency and added a new row in a list of items that will indicate the count of requests.

If we run the application, we will see at the top the current count of requests made to the server. If we kill and reopen the application, then we will see that count increase, which shows how it will survive the application being stopped by the user or killed by the operating system.

In this section, we have analyzed another common way of persisting data on an Android device through the DataStore library. We also observed how easy it is for the DataStore library to be integrated with flows and other libraries such as Room and Retrofit.

Summary

In this chapter, we have looked at how we can load and persist data in Android and the rules we must follow for threading. We first analyzed how we can load data asynchronously and focused on coroutines and flows, for which we have done simple exercises for performing asynchronous operations on different threads and updating the UI on the main thread. We then studied how to load data from the internet using OkHttp and Retrofit, and followed this up with how to persist data using Room and DataStore and how we can integrate all of these with coroutines and flows. We highlighted the usage of these libraries in exercises, and we also showed how they can be integrated with coroutines and flows. The integration of different flows of data was combined in the **`ViewModel`** class, in which we loaded the network data and inserted it into the local database. This is generally not a good approach, and we will expand on how we can improve this in future chapters.

In the next chapter, we will look at how we can present data to the user and the libraries and frameworks we can use to achieve this.