

# Chapter 4: Handling Async Operations with Coroutines

In this chapter, we're focusing on another library that, although is not in the Jetpack library suite, is essential for writing solid applications: **Kotlin coroutines**.

Coroutines represent a more convenient way of handling async work and concurrency jobs on Android.

In this chapter, we will study how we can replace callbacks with coroutines in our Restaurants application. In the first section, *Introducing Kotlin coroutines*, we will gain a better understanding of what coroutines are, how they work, and why we need them in our apps.

In the next section, *Exploring the basic elements of coroutines*, we will explore the core elements of coroutines, and we will understand how to use them to handle asynchronous work more concisely.

Finally, in the *Using coroutines for async work* section, we will implement coroutines in our Restaurants application and let them handle the network requests. Additionally, we will add error handling and integrate some of the best practices when working with coroutines in Android apps.

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

- Introducing Kotlin coroutines
- Exploring the basic elements of coroutines
- Using coroutines for async work

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

# Technical requirements

Building Compose-based Android projects with coroutines usually requires your day-to-day tools. However, to follow along with this chapter 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 a newer plugin, installed in Android Studio
- The Restaurants app code from the previous chapter.

The starting point for this chapter is represented by the Restaurants application developed in the previous chapter. If you haven't followed the implementation from the previous chapter, access the starting point for this chapter by navigating to the **Chapter\_03** directory of the repository and importing the Android project entitled **chapter\_3\_restaurants\_app**.

To access the solution code for this chapter, navigate to the **Chapter\_04** directory:

[https://github.com/PacktPublishing/Kickstart-Modern-Android-Development-with-Jetpack-and-Kotlin/tree/main/Chapter\\_04/chapter\\_4\\_restaurants\\_app](https://github.com/PacktPublishing/Kickstart-Modern-Android-Development-with-Jetpack-and-Kotlin/tree/main/Chapter_04/chapter_4_restaurants_app).

## Introducing Kotlin coroutines

**Coroutines** are part of the Kotlin API. They introduce a new and easier way of handling async work and concurrency jobs.

Often, with Android, we need to run or execute different tasks behind the scenes. In the meantime, we don't want to block the main thread of the application and get an unresponsive UI.

To mitigate this issue, coroutines allow you to execute async work much easier while providing main-thread safety for your Android apps. You can use the Coroutines API by launching one **coroutine**, or more, depending on your needs.

In this section, we will cover three essential questions about the Coroutines API that derive from what we stated earlier:

- What is a coroutine?
- What are the features and advantages of coroutines?
- How do coroutines work?

Let's jump in!

## What is a coroutine?

A coroutine is a concurrency design pattern for async work. A *coroutine represents an instance of suspendable computation*.

In other words, coroutines are sequences or blocks of code that represent a computational task that can be suspended. We call them **suspendable** because coroutines can be suspended and resumed mid-execution, which makes them efficient for concurrent tasks.

When comparing coroutines with threads, we can say the following:

- A coroutine is a lightweight version of a thread but not a thread. Coroutines are light because creating coroutines doesn't allocate new threads – typically, coroutines use predefined thread pools.
- Like threads, coroutines can run in parallel, wait for each other, and communicate.
- Unlike threads, coroutines are very cheap: we can create thousands of them and pay very few penalties in terms of performance.

Next, let's understand the purpose behind coroutines a bit better.

## The features and advantages of coroutines

By now, we know that on Android, coroutines can help us to move long-running async work from the main thread into a separate thread.

Essentially, coroutines have two primary possible usages:

- For handling async work
- For handling multithreading

In this chapter, we will only cover how to correctly handle async work with coroutines in Android apps.

However, before we try to understand how to do that with coroutines, let's explore the advantages that coroutines bring over other alternatives that we've used in the past: **AsyncTask** classes, callbacks, and reactive frameworks. A coroutine is described as the following:

- **Lightweight:** We can launch many coroutines on a single thread. Coroutines support execution suspension on the thread as opposed to blocking it, resulting in less memory overhead. Additionally, a coroutine is not always bound to a specific thread – it might start its execution on one thread and yield the result on a different one.
- **Easily cancelable:** When canceling the parent coroutine, any children coroutines that were launched within the same scope will be canceled. If you have launched multiple coroutines that run operations concurrently, cancelation is straightforward and applies to the entire affected coroutine hierarchy; therefore, this eliminates any potential memory leaks.
- **Easily integrated with Jetpack libraries:** By providing a suite of extensions. For example, coroutines provide custom scopes for many common Android components such as **Activity**, **Fragment**, **ViewModel**, and more. This means that you can launch coroutines safely from these components, as they will be canceled automatically when different lifecycle events occur, so you don't have to worry about memory leaks.

## NOTE

*We have mentioned the word scope several times, and I promise that we will explain it later. Until then, you can think of the coroutine scope as an entity that controls the lifetime of launched coroutines.*

Now we have an idea of the features of coroutines. Yet, to better understand their purpose, first, we need to understand why we should offload async work from the main thread to a separate worker thread.

## How do coroutines work?

In Android runtime, the main thread is responsible for two things:

- Drawing the UI of the application on the screen
- Updating the UI upon user interactions

Simplistically viewed, the main thread calls a drawing method on the screen canvas. This method might be familiar to you as the `onDraw()` method, and we can assume that for your device to render UI at 60 frames per second, the Android Runtime will call this method roughly every 16 milliseconds.

If, for some reason, we execute heavy async work on the main thread, the application might freeze or stutter. This happens because the main thread was busy serving our async work; therefore, it missed several `onDraw()` calls that could have updated the UI and prevented the freezing effect.

Let's say that we need to make a network request to our server. This operation might take time because we must wait for a response, which depends on the web API's speed and the user's connectivity. Let's imagine that such a method is named `getNetworkResponse()` and we are calling it from the main thread:

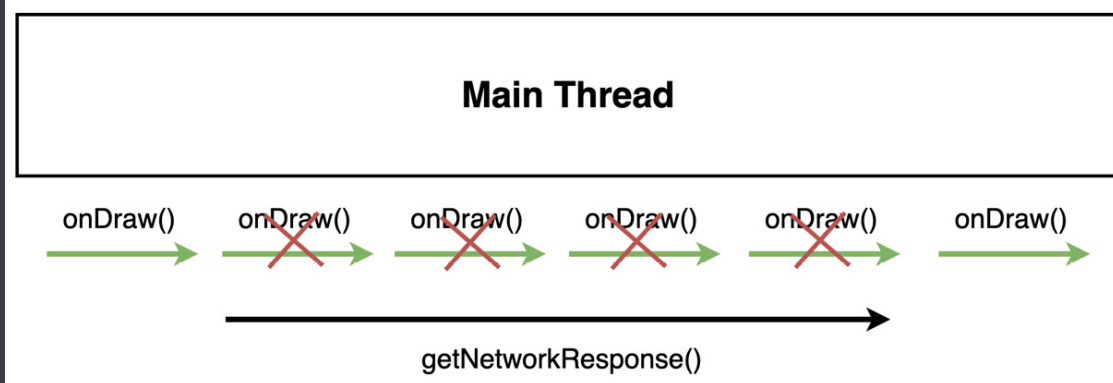


Figure 4.1 – Blocking the main thread with async work

From the time it launched the network request, the main thread kept waiting for a response and couldn't do anything in the meantime. We can see that several `onDraw()` calls were missed because the main thread was busy executing our `getNetworkResponse()` method call and waiting for a result.

To mitigate this issue, we've used many mechanisms in the past. Yet, coroutines are much easier to use and work perfectly with the Android ecosystem. So, it's time to see how they can enable us to execute async work:

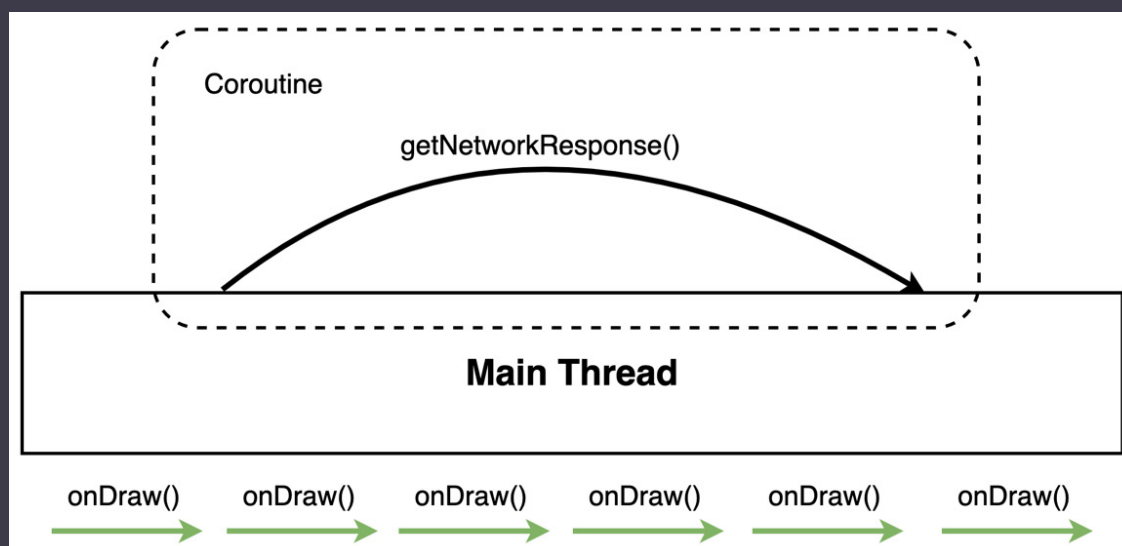


Figure 4.2 – Executing async work on a different thread by using a coroutine

With coroutines, we can offload any nasty blocking calls – such as the `getNetworkResponse()` method call – from the main thread onto a

coroutine.

The coroutine works on a separate thread and is in charge of executing the network request and waiting for the response. This way, the main thread is not blocked, and no `onDraw()` calls are missed; therefore, we avoid getting any freezing screen effects.

Now that we have a basic understanding of how coroutines work, it's time to explore the components that coroutines are based on.

## Exploring the basic elements of coroutines

A very simplistic approach for getting async work done with coroutines could be expressed as follows: first, define the suspended functions and then create coroutines that execute the suspended functions.

Yet, we're not only unsure what suspending functions look like, but we also don't know how to allow coroutines to perform asynchronous work for us.

Let's take things, step by step, and start with the two essential actions that we need to execute async work with coroutines:

- Creating suspending functions
- Launching coroutines

All of these terms make little sense now, so let's address this, starting with suspending functions!

### Creating suspending functions

The first thing that we need in order to work with coroutines is to define a suspending function where the blocking task resides.

A **suspending** function is a special function that can be paused (suspended) and resumed at some later point in time. This allows us to execute long-running jobs while the function is suspended and, finally, resume it when the work is complete.

Our regular function calls within our code are mostly executed synchronously on the main thread. Essentially, suspending functions allow us to execute jobs asynchronously in the background without blocking the thread where those functions are called from.

Let's say that we need to save some details about a user to a local database. This operation takes time, so we need to display an animation until it finishes:

```
fun saveDetails(user: User) {  
    startAnimation()  
    database.storeUser(user)  
    stopAnimation()  
}
```

If this operation is called on the main thread, the animation will freeze for a few hundreds of milliseconds while the user's details are saved.

Take a closer look at the code presented earlier and ask yourself the following: *which method call should be suspendable?*

Since the `storeUser()` method takes a while to finish, we want this method to be a suspending function because this function should be paused until the user's details are saved and then resumed when the job is done. This ensures that we do not block the main thread or freeze the animation.

Yet, how can we make the `storeUser()` method a suspending function?

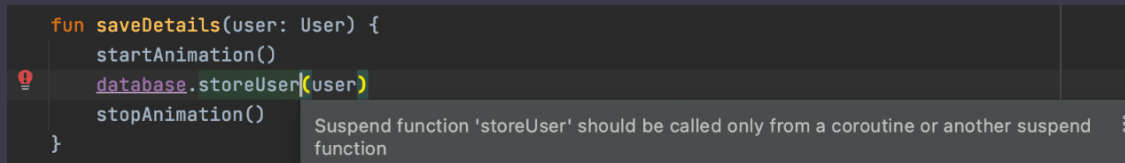
A suspending function is a regular function that is marked with the **suspend** keyword:



```
suspend fun storeUser(user: User) {  
    // blocking action  
}
```

We know that the `storeUser()` method saves details to a database, which takes a good while. So, in order to prevent this job from blocking the UI, we've marked the method with an additional `suspend` keyword.

However, if we mark a method with the `suspend` keyword, trying to call it in our code results in a compilation error:



```
fun saveDetails(user: User) {  
    startAnimation()  
    database.storeUser(user)  
    stopAnimation()  
}
```

Suspend function 'storeUser' should be called only from a coroutine or another suspend function

Figure 4.3 – Calling suspending functions from regular functions results in a compilation error

Suspending functions can only be called from inside a coroutine or from inside another suspending function. Instead of calling our `storeUser()` suspending method from a regular method, let's create a coroutine and call it from there.

## Launching coroutines

To execute a suspend function, first, we need to create and launch a coroutine. To do that, we need to call a coroutine builder on a coroutine scope:

```
fun saveDetails(user: User) {  
    GlobalScope.launch(Dispatchers.IO) {  
        startAnimation()  
        database.storeUser(user)  
        stopAnimation()  
    }  
}
```

We have just launched our first coroutine and called our suspending function inside it! Let's break down what just happened:

- We've used a **GlobalScope** coroutine scope, which manages the coroutines that are launched within it.
- In the coroutine scope, we called the **launch()** coroutine builder to create a coroutine.
- Then, we passed the **Dispatchers.IO** dispatcher to the coroutine builder. In this case, we want to save the user details inside the database on a thread reserved for I/O operations.
- Inside the block that the **launch()** coroutine builder has provided us with, we call our **storeUser()** suspending function.

Now we have successfully moved our blocking work away from the main thread to a worker thread. Therefore, we have made sure that the UI will not be blocked, and the animation will run smoothly.

However, now that we have implemented suspending work in our **saveDetails()** method, you might be wondering what the order of function calls within this method will be.

To better understand how the regular synchronous world blends with the suspending world, let's add some logs to our previous code snippet:

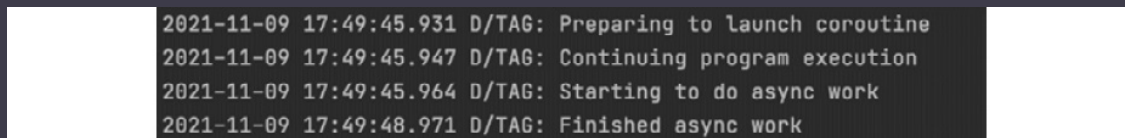
```
fun saveDetails(user: User) {  
    Log.d("TAG", "Preparing to launch coroutine")  
    GlobalScope.launch(Dispatchers.IO) {  
        startAnimation()  
        Log.d("TAG", "Starting to do async work")  
        database.storeUser(user)  
        Log.d("TAG", "Finished async work")  
        stopAnimation()  
    }  
    Log.d("TAG", "Continuing program execution")  
}
```

Remember, the only suspending function in this block of code that will take some time to compute is `database.storeUser()`. Now, let's imagine that we have run the preceding piece of code.

### EXERCISE

*Before checking the following output, try to think about the order of the logs yourself. What do you expect the order of function calls to be?*

Let's see the output:



```
2021-11-09 17:49:45.931 D/TAG: Preparing to launch coroutine
2021-11-09 17:49:45.947 D/TAG: Continuing program execution
2021-11-09 17:49:45.964 D/TAG: Starting to do async work
2021-11-09 17:49:48.971 D/TAG: Finished async work
```

Figure 4.4 – The output order of regular and suspending functions

The order of the function calls is a bit out of order, but it is definitely correct. Let's see what happened:

1. First, the log function with the **Preparing to launch coroutine** message was called. This method call was done on the main (UI) thread.
2. Even though up next, we launched the coroutine, we can see that the second log function called was the last one in our code: **Continuing program execution**.

This is because the coroutine is a bridge to the suspending world, so every function call from the coroutine will be run on a different thread from the main thread. More precisely, the operation of switching from the main thread to `Dispatchers.IO` will take some time. This means that all of these methods inside the coroutine will be executed after the method call outside of the coroutine.

3. The next log function call is with the **Starting to do async work** message. This method is called inside the coroutine on a thread reserved for I/O operations. This log marks the start of execution for all suspending work.

4. Finally, after all of the blocking work from the `database.storeUser()` suspending function has been finished, the last log function call with the **Finished async work** message is called. This log marks the end of the coroutine execution.

Now that we've understood how the regular world blends with the suspended world in terms of function calls, there are still many terms and concepts that have been thrown at you. Mainly, you might be wondering the following:

- What is a coroutine scope?
- What's a coroutine dispatcher?
- What's a coroutine builder?

Let's clarify these concepts, starting with coroutine scopes.

## Coroutine scopes

Essentially, coroutines run in **coroutine scopes**. To start a coroutine, first, you need a coroutine scope because it tracks all of the coroutines launched inside it and has the ability to cancel them. This way, you can control how long the coroutines should live and when they should be canceled.

A coroutine scope contains a **CoroutineContext** object, which defines the context in which the coroutine runs. In the previous example, we used a predefined scope, **GlobalScope**, but you can also define a custom scope by constructing a **CoroutineContext** object and passing it to a **CoroutineScope()** function, as follows:

```
val job = Job()
val myScope = CoroutineScope(context = job +
    Dispatchers.IO)
```

The **CoroutineScope()** function expects a **CoroutineContext** object passed to its **context** parameter and knows how to build one out of the box. It

does this by receiving elements with a special `plus` operator and then constructing the context behind the scenes.

Most of the time, the two most important elements to construct a **CoroutineContext** object are the ones that we just passed:

- A **Job** object: This represents a cancelable component that controls the lifecycle of a coroutine launched in a specific scope. When a job is canceled, the job will cancel the coroutine it manages. For example, if we have defined a **job** object and a custom **myScope** object inside an **Activity** class, a good place to cancel the coroutine would be in the **onDestroy()** callback by calling the **cancel()** method on the **job** object:

```
override fun onDestroy() {  
    super.onDestroy()  
    job.cancel()  
}
```

By doing this, we've ensured that our async work done within our coroutine, which uses the **myScope** scope, will stop when the activity has been destroyed and will not cause any memory leaks.

- A **Dispatcher** object: Marking a method as suspended provides no details about the thread pool it should run on. So, by passing a **Dispatcher** object to the **CoroutineScope** constructor, we can make sure that all suspended functions called in the coroutine that use this scope will default to the specified **Dispatcher** object. In our example, all coroutines launched in **myScope** will run their work, by default, in the **Dispatchers.IO** thread pool and will not block the UI.

Note that the **CoroutineContext** object can also contain an exception handler object, which we will define later on.

Apart from the custom scopes that you can define, as we did earlier, you can use predefined coroutine scopes that are bound to a certain lifecycle

component. In such cases, you will no longer need to define a scope with a job or to manually cancel the coroutine scope:

- **GlobalScope**: This allows the coroutines to live as long as the application is alive. In the previous example, we used this scope for simplicity, but **GlobalScope** should be avoided since the work launched within this coroutine scope is only canceled when the application has been destroyed. Using this scope in a component that has a narrower lifecycle than the application – such as an **Activity** component, might allow the coroutine to outlive that component's lifecycle and produce memory leaks.
- **lifecycleScope**: This scopes coroutines to the lifecycle of a **LifecycleOwner** instance such as an **Activity** component or a **Fragment** component. We can use the **lifecycleScope** scope defined in the Jetpack KTX extensions package:

```
class UserFragment : Fragment() {  
    ...  
    fun saveDetails(user: User) {  
        lifecycleScope.launch(Dispatchers.IO) {  
            startAnimation()  
            database.storeUser(user)  
            stopAnimation()  
        }  
    }  
}
```

By launching coroutines within this context, we ensure that if the **Fragment** component gets destroyed, the coroutine scope will automatically be canceled; therefore, this will also cancel our coroutine.

- **viewModelScope**: To scope our coroutines to live as long as the **ViewModel** component does, we can use the predefined **viewModelScope** scope:

```
class UserViewModel: ViewModel() {  
    fun saveDetails(user: User) {
```

```
        // do some work
        viewModelScope.launch(Dispatchers.IO) {
            database.storeUser(user)
        }
        // do some other work
    }
}
```

By launching coroutines within this context, we ensure that if the **ViewModel** component gets cleared, the coroutine scope will cancel its work – in other words, it will automatically cancel our coroutine.

- **rememberCoroutineScope**: To scope a coroutine to the composition cycle of a composable function, we can use the predefined **rememberCoroutineScope** scope:

```
@Composable
fun UserComposable() {
    val scope = rememberCoroutineScope()
    LaunchedEffect(key1 = "save_user") {
        scope.launch(Dispatchers.IO) {
            viewModel.saveUser()
        }
    }
}
```

Therefore, a coroutine's lifecycle is bound to the composition cycle of **UserComposable**. This means that when **UserComposable** leaves the composition, the scope will be automatically canceled, thereby preventing the coroutine from outliving the composition lifecycle of its parent composable.

Since we want the coroutine to be launched only once upon composition and not at every recomposition, we wrapped the coroutine with a **LaunchedEffect** composable.

Now that we covered what coroutine scopes are and how they allow us to control the lifetime of coroutines, it's time to better understand what dispatchers are.

## Dispatchers

A **CoroutineDispatcher** object allows us to configure what thread pool our work should be executed on. The point of coroutines is to help us move blocking work away from the main thread. So, somehow, we need to instruct the coroutines what threads to use for the work that we pass to them.

To do that, we need to configure the **CoroutineContext** object of the coroutines to set a specific dispatcher. In fact, when we covered coroutine scopes, we've explained how **CoroutineContext** is defined by a job and a dispatcher.

When creating custom scopes, we can specify the default dispatcher right when we instantiate the scope, just as we did previously:

```
val myScope = CoroutineScope(context = job +  
    Dispatchers.IO)
```

In this case, the default dispatcher of **myScope** is **Dispatchers.IO**. This means that whatever suspending work we pass to the coroutines that are launched with **myScope**, the work will be moved to a special thread pool for I/O background work.

In the case of predefined coroutine scopes, such as with **lifecycleScope**, **viewModelScope**, or **rememberCoroutineScope**, we can specify the desired default dispatcher when starting our coroutine:

```
scope.launch(Dispatchers.IO) {  
    viewModel.saveUser()  
}
```

We start coroutines with coroutine builders such as **launch** or **async**, which we will cover in the next section. Until then, we need to under-



stand that when launching a coroutine, we can also modify the **CoroutineContext** object of the coroutine by specifying a **CoroutineDispatcher** object.

Now we've used **Dispatchers.IO** as a dispatcher throughout our examples. But are there any other dispatchers that are of use to us?

**Dispatchers.IO** is a dispatcher offered by the Coroutines API, but in addition to this, coroutines offer other dispatchers too. Let's list the most notable dispatchers as follows:

- **Dispatchers.Main**: This dispatches work to the main thread on Android. It is ideal for light work (which doesn't block the UI) or actual UI function calls and interactions.
- **Dispatchers.IO**: This dispatches blocking work to a background thread pool that specializes in handling disk-heavy or network-heavy operations. This dispatcher should be specified for suspending work on local databases or executing network requests.
- **Dispatchers.Default**: This dispatches blocking work to a background thread pool that specializes in CPU-intensive tasks, such as sorting long lists, parsing JSON, and more.

In the previous examples, we set a specific dispatcher of **Dispatchers.IO** for the **CoroutineContext** object of the coroutines launched, ensuring that suspended work will be dispatched by this specific dispatcher.

But we've made a critical mistake! Let's take a look at the code again:

```
class UserFragment : Fragment() {  
    ...  
    fun saveDetails(user: User) {  
        lifecycleScope.launch(Dispatchers.IO) {  
            startAnimation()  
            database.storeUser(user)  
            stopAnimation()  
        }  
    }  
}
```

```
    }  
  }  
}
```

The main issue with this code is that the `startAnimation()` and `stopAnimation()` functions are probably not even suspending functions, as they interact with the UI.

We wanted to run our `database.storeUser()` blocking work on a background thread, so we specified the `Dispatchers.IO` dispatcher to the `CoroutineContext` object. But this means that all the rest of the code in the coroutine block (that is, the `startAnimation()` and `stopAnimation()` function calls) will be dispatched to a thread pool intended for background work instead of being dispatched to the main thread.

To have more fine-grained control regarding what threads our functions are being dispatched to, coroutines allow us to control the dispatcher by using the `withContext` block, which creates a block of code that can run on a different dispatcher.

Since `startAnimation()` and `stopAnimation()` have to work on the main thread, let's refactor our example.

Let's launch our coroutine with the default dispatcher of `Dispatchers.Main`, and then wrap our work, which has to be run on a background thread (the `database.storeUser(user)` suspending function), with a `withContext` block:

```
fun saveDetails(user: User) {  
    lifecycleScope.launch(Dispatchers.Main) {  
        startAnimation()  
        withContext(Dispatchers.IO) {  
            database.storeUser(user)  
        }  
        stopAnimation()  
    }  
}
```

```
}
```

The `withContext` function allows us to define a more granular `CoroutineContext` object for the block that it exposes. In our case, we had to pass the `Dispatchers.IO` dispatcher to make sure our blocking work with the database will run on the background thread instead of being dispatched to the main thread.

In other words, our coroutine will have all its work dispatched to the `Dispatchers.Main` dispatcher, unless you define another more granular context that has its own `CoroutineDispatcher` set.

Now we've covered how to use dispatchers and how to ensure more granular control over how our work is dispatched to different threads. However, we haven't covered what the `launch { }` block means. Let's do that next.

## Coroutine builders

**Coroutine builders** (such as `launch`) are extension functions on `CoroutineScope` and allow us to create and start coroutines. Essentially, they are a bridge between the normal synchronous world with regular functions and the suspending world with suspending functions.

Since we can't call suspending functions inside regular functions, a coroutine builder method executed on the `CoroutineScope` object creates a scoped coroutine that provides us with a block of code where we can call our suspending functions. Without scopes, we cannot create coroutines – which is good since this practice helps to prevent memory leaks.

We can use three builder functions to create coroutines:

- **launch**: This starts a coroutine that runs concurrently with the rest of the code. Coroutines started with `launch` won't return the result to the caller – instead, all of the suspending functions will run sequentially inside the block that `launch` exposes. It's our job to get the result from the suspending functions and then interact with that result:

```
fun getUser() {  
    lifecycleScope.launch(Dispatchers.IO) {  
        val user = database.getUser()  
        // show details to UI  
    }  
}
```

Most of the time, if you don't need concurrent work, **launch** is the go-to option for starting coroutines since it allows you to run your suspending work inside the block of code provided and doesn't care about anything else.

If no dispatcher is specified in the coroutine builder, the dispatcher that is going to be used is the dispatcher provided by default by the **CoroutineScope** used to start the coroutine. In our case, if we wouldn't have specified a dispatcher, our coroutine launched with the **launch** coroutine builder will have used the **Dispatchers.Main** dispatcher defined by default by **lifecycleScope**.

Apart from **lifecycleScope**, **viewModelScope** also provides the same predefined dispatcher of **Dispatchers.Main**. **GlobalScope** on the other hand, defaults to **Dispatchers.Default** if no dispatcher was provided to the coroutine builder.

- **async**: This starts a new coroutine, and it allows you to return the result as a **Deferred<T>** object, where **T** is your expected data type. The deferred object is a promise that your result, **T**, will be returned in the future. To start the coroutine and get a result, you need to call the suspending function, **await**, which blocks the calling thread:

```
lifecycleScope.launch(Dispatchers.IO) {  
    val deferredAudio: Deferred<Audio> =  
        async { convertTextToSpeech(title) }  
    val titleAudio = deferredAudio.await()  
    playSound(titleAudio)  
}
```

We can't use **async** in a normal function as it has to call the **await** suspending function to get the result. To fix that, first, we've created a parent coroutine with **launch** and started the child coroutine with **async** inside it. This means the child coroutine that was started with **async** inherits its **CoroutineContext** object from the parent coroutine that was started with **launch**.

With **async**, we can get the results of the concurrent work in one place. Where the **async** coroutine builder shines (and where it's recommended to be used) is in tasks with parallel execution where results are required.

Let's say that we need to simultaneously convert two pieces of text into speech and then play both results at the same time:

```
lifecycleScope.launch(Dispatchers.IO) {  
    val deferredTitleAudio: Deferred<Audio> =  
        async { convertTextToSpeech(title) }  
    val deferredSubtitleAudio: Deferred<Audio> =  
        async { convertTextToSpeech(subtitle) }  
    playSounds(  
        deferredTitleAudio.await(),  
        deferredSubtitleAudio.await()  
    )  
}
```

In this particular example, both the resulting **deferredTitleAudio** and **deferredSubtitleAudio** tasks will run in parallel.

Since our Restaurants application hasn't featured concurrent work until now, we won't go any deeper in terms of concurrency topics.

- **runBlocking**: This starts a coroutine that blocks the current thread on which it is invoked until the coroutine has been completed. This builder should be avoided for async work within our app since creating threads and blocking them is less efficient. However, this coroutine builder can be used for unit tests.

Now that we have covered the basics of coroutines, it's high time we implement coroutines in our Restaurants application!

## Using coroutines for async work

The first thing that we have to do is identify the async/heavy work that we have done in our Restaurants application.

Without looking at the code, we know that our app retrieves a list of restaurants from the server. It does that by initiating a network request with Retrofit and then waits for a response. This action qualifies as an async job because we don't want to block the main (UI) thread while the app waits for the network response to arrive.

If we check out the `RestaurantsViewModel` class, we can identify that the `getRestaurants()` method is the one place in our application where heavy blocking work is happening:

```
private fun getRestaurants() {
    restaurantsCall = restInterface.getRestaurants()
    restaurantsCall.enqueue(object : Callback
        <List<Restaurant>> {
        override fun onResponse(...) {
            response.body()?.let { restaurants ->
... }
        }
        override fun onFailure(...) {
            t.printStackTrace()
        }
    })
}
```

When we implemented the network request, we used Retrofit's `enqueue()` method to which we passed a `Callback` object where we could wait for the result without blocking the main thread.

To simplify the way we handle this async operation of getting the restaurants from the server, we will implement coroutines. This will allow us to ditch callbacks and make our code more concise.

In this section, we will cover two main steps:

- Implementing coroutines instead of callbacks
- Improving the way our app works with coroutines

Let's get started!

## Implementing coroutines instead of callbacks

To handle async work with coroutines, we need to do the following:

- Define our async work in a suspending function.
- Next, create a coroutine and call the suspending function inside it to obtain the result asynchronously.

Enough with the theory, it's time to code! Perform the following steps:

1. Inside the **RestaurantsApiService** interface, add the **suspend** keyword to the **getRestaurants()** method and replace the **Call<List<Restaurant>>** return type of the method with **List<Restaurant>**:

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

Retrofit supports coroutines out of the box for network requests. This means that we can mark any method in our Retrofit interface with the **suspend** keyword; therefore, we can transform the network requests to suspending work that isn't blocking the main thread of the application.

Because of this, the `Call<T>` return type is redundant. We no longer need Retrofit to return a `Call` object on which we would normally enqueue a `Callback` object to listen for the response – all of this will be handled by the Coroutines API.

2. Since we will no longer receive a `Call` object from Retrofit, we will also not need the `Callback` object in our `RestaurantsViewModel` class. Clean up the `RestaurantsViewModel` component:

1. Remove the `restaurantsCall: Call<List<Restaurant>` member variable.
2. Remove the `restaurantsCall.cancel()` method call inside the `onCleared()` callback.
3. Remove the entire body of the `getRestaurants()` method.

3. Inside the `getRestaurants()` method, call the `restInterface.getRestaurants()` suspending function and store the result in a `restaurants` variable:

```
private fun getRestaurants() {  
    val restaurants =  
    restInterface.getRestaurants()  
}
```

The IDE will throw an error telling us that we cannot call the `restInterface.getRestaurants()` suspending function from the regular `getRestaurants()` function within the `ViewModel` component.

To fix this, we must create a coroutine, launch it, and call the suspending function there.

4. Before creating a coroutine, we need to create a `CoroutineScope` object. Inside the `ViewModel` component, define a member variable of type `Job` and another of type `CoroutineScope`, just as we learned earlier:

```
class RestaurantsViewModel(...): ViewModel() {  
    private var restInterface:  
    RestaurantsApiService  
    val state = mutableStateOf(...)
```



```
    val job = Job()
    private val scope = CoroutineScope(job +
        Dispatchers.IO)
    ...
}
```

The **job** variable is the handle that will allow us to cancel the coroutine scope, while the **scope** variable will ensure we keep track of the coroutines that are going to be launched with it.

Since the network request is a heavy blocking operation, we want its suspending work to be executed on the **IO** thread pool to avoid blocking the main thread, so we specified the **Dispatchers.IO** dispatcher for our **scope** object.

5. Inside the **onCleared()** callback method, call the **cancel()** method in the newly created job variable:

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

By calling **cancel()** on our **job** variable, we ensure that if the **RestaurantsViewModel** component is destroyed (for example, in scenarios where the user navigates to a different screen) the coroutine **scope** object will be canceled through its **job** object reference. Effectively, this will cancel any suspending work and prevent the coroutine from causing a memory leak.

6. Inside the **getRestaurants()** method in our **ViewModel** component, create a coroutine by calling **launch** on the previously defined **scope** object, and inside that body exposed by the coroutine add the existing code where we obtain the restaurants:

```
private fun getRestaurants() {
    scope.launch {
```

```
        val restaurants =  
restInterface.getRestaurants()  
    }  
}
```

Success! We have launched a coroutine that executes our suspending work of obtaining the restaurants from the server.

7. Next, add the initial code to update our **State** object with the newly received restaurants so that the Compose UI displays them:

```
scope.launch {  
    val restaurants =  
restInterface.getRestaurants()  
    state.value = restaurants.restoreSelections()  
}
```

However, this approach is flawed. Can you point out why?

Well, we are updating the UI on an incorrect thread. Our **scope** is defined to run the coroutine on a thread from the **Dispatchers.IO** thread pool, but updating the UI should happen on the Main thread.

8. Inside the **getRestaurants()** method, wrap the line of code where the Compose **State** object is updated with a **withContext** block that specifies the **Dispatchers.Main** dispatcher:

```
scope.launch {  
    val restaurants =  
restInterface.getRestaurants()  
    withContext(Dispatchers.Main) {  
        state.value =  
restaurants.restoreSelections()  
    }  
}
```

By doing this, we ensure that while heavy work is being done on the background threads, the UI is updated from the main thread.

We have now successfully implemented coroutines in our app. We have defined a scope and created a coroutine where we executed our suspending work: a network request.

9. You can now **Run** the application and notice that on the outside, the behavior of the app hasn't changed. However, behind the scenes, our async work was done with the help of coroutines in a more elegant manner than before.

Even so, there are a few things that could be improved. Let's tackle those next.

## Improving the way our app works with coroutines

Our app uses a coroutine to move heavy work from the main thread to specialized threads.

However, if we think about our particular implementation, we can find some ways to improve our coroutine-related code:

- Use predefined scopes as opposed to custom scopes.
- Add error handling.
- Make sure that every **suspend** function is safe to be called on any **Dispatcher** object.

Let's start with the fun one: replacing our custom scope with a predefined one!

### Using predefined scopes as opposed to custom scopes

In our current implementation, we've defined a custom **CoroutineScope** object that will make sure that its coroutines will live as long as the **RestaurantsViewModel** instance. To achieve this, we pass a **Job** object to our **CoroutineScope** builder and cancel it when the **ViewModel** component is destroyed: on the **onCleared()** callback method.

Now, remember that coroutines are well integrated with the Jetpack libraries, and when we define scopes, we also talk about predefined scopes such as **lifecycleScope**, **viewModelScope**, and more. These scopes make sure that their coroutines live as long as the component they are bound to, for example, **lifecycleScope** is bound to a **Fragment** or **Activity** component.

### NOTE

*Whenever you are launching a coroutine inside components such as **Activity**, **Fragment**, **ViewModel**, or even composable functions, remember that instead of creating and managing your own **CoroutineScope** object, you can use the predefined ones that take care of canceling coroutines automatically. By using predefined scopes, you can better avoid memory leaks as any suspending work is cancelled when needed.*

In our scenario, we can simplify our code and replace our custom **CoroutineScope** object with the **viewModelScope** one. Behind the scenes, this predefined scope will take care of canceling all of the coroutines launched with it when its parent **ViewModel** instance has been cleared or destroyed.

Let's do that now:

1. Inside the **getRestaurants()** method of the **RestaurantsViewModel** class, replace **scope** with **viewModelScope**:

```
private fun getRestaurants() {  
    viewModelScope.launch {  
        val restaurants = ...
```

```
        ...  
    }  
}
```

2. Since we will no longer use our **scope** object, we need to make sure that our coroutine will run the suspending work in the background, just as it did with the previous scope. Pass a **Dispatchers.IO** dispatcher to the **launch** method:

```
viewModelScope.launch(Dispatchers.IO) {  
    val restaurants =  
    restInterface.getRestaurants()  
    withContext(Dispatchers.Main) {  
        state.value =  
        restaurants.restoreSelections()  
    }  
}
```

Usually, the **launch** coroutine builder inherits **CoroutineContext** from its parent coroutine. In our particular case though, if no dispatcher is specified, coroutines launched with **viewModelScope** will default to using **Dispatchers.Main**.

However, we want our network request to be executed on a background thread from the specialized I/O thread pool, so we passed an initial **CoroutineContext** object with a **Dispatchers.IO** dispatcher to our **launch** call.

3. Remove the **onCleared()** callback method entirely from the **ViewModel** class. We will no longer need to cancel our coroutine **scope** from a **job** object because **viewModelScope** takes care of that for us.
4. Remove the **job** and **scope** member variables from the **RestaurantsViewModel** class.
5. You can now **Run** the application and again notice that on the outside, the behavior of the app hasn't changed. Our code now works the same but is greatly simplified because we used a predefined scope instead of handling everything by ourselves.

Next, we must re-add error handling to our project. However, this time, we will do it in the context of coroutines.

## Adding error handling

In the previous implementation with callbacks, we received an error callback from Retrofit. However, with coroutines, it appears that since our suspending function returns `List<Restaurant>>`, there is no room for error.

Indeed, we are not handling any error that could be thrown. For example, if you try to launch the application without internet right now, Retrofit will throw a `Throwable` object, which, in turn, will crash our app with a similar error as follows:

```
E/AndroidRuntime: FATAL EXCEPTION: DefaultDispatcher-
worker-1
```

To handle errors, we can simply wrap suspending function calls in a `try catch` block:

```
viewModelScope.launch(Dispatchers.IO) {
    try {
        val restaurants =
            restInterface.getRestaurants()
        // show restaurants
    } catch (e: Exception) {
        e.printStackTrace()
    }
}
```

The preceding approach is fine, but the code becomes less concise because of another level of nesting. Additionally, to better support a single point of error handling, coroutines allow you to pass a `CoroutineExceptionHandler` object to the context of your `CoroutineScope` object:

```

Creates a CoroutineExceptionHandler instance.
Params: handler - a function which handles exception thrown by a coroutine
@Suppress(...names: "FunctionName")
public inline fun CoroutineExceptionHandler(crossinline handler: (CoroutineContext, Throwable) → Unit): CoroutineExceptionHandler =
    object : AbstractCoroutineContextElement(CoroutineExceptionHandler), CoroutineExceptionHandler {
        override fun handleException(context: CoroutineContext, exception: Throwable) =
            handler.invoke(context, exception)
    }

```

Figure 4.5 – The signature of CoroutineExceptionHandler

The **CoroutineExceptionHandler** object allows us to handle errors thrown by any coroutine launched within a **CoroutineScope** object, no matter how nested it might be. This handler gives us access to a function that exposes the **CoroutineContext** object and the **Throwable** object thrown in this particular context.

Let's add such a handler to the **RestaurantsViewModel** class. Perform the following steps:

1. Define an **errorHandler** member variable of type **CoroutineExceptionHandler** and print the stack trace of the **exception: Throwable** parameter:

```

class RestaurantsViewModel() : ViewModel() {
    ...
    private val errorHandler =
        CoroutineExceptionHandler { _, exception ->
            exception.printStackTrace()
        }
    ...
}

```

We're not interested in the first parameter of type **CoroutineContext**, so we named it with an underscore, **\_**.

2. Inside the **getRestaurants()** method, pass the **errorHandler** variable to the **launch** block using the **+** operator:

```

private fun getRestaurants() {
    viewModelScope.launch(Dispatchers.IO +
        errorHandler) {

```

```
        ...  
    }  
}
```

By passing our `errorHandler` variable to the `launch` method, we make sure that the `CoroutineContext` object of this coroutine sets this `CoroutineExceptionHandler`, which will allow us to handle errors inside our handler.

3. Try running the app again without the internet.

Now the app shouldn't crash because the `errorHandler` variable will catch the `Throwable` object thrown by Retrofit and allow us to print its stack trace.

#### NOTE

*As an improvement, try to find a way of notifying the UI that an error has occurred, thereby informing the user of what just happened.*

We are now handling errors with coroutines, so it's time to move to the last point of improvement – handling the switch of dispatchers correctly.

### Making sure that every suspending function is safe to be called on any dispatcher

When defining suspending functions, a good practice is to make sure that every suspending function can be called on any `Dispatcher` object. This way, the caller (in our case, the coroutine) doesn't have to worry about what thread will be needed to execute the suspending function.

Let's analyze our code with the coroutine:

```
private fun getRestaurants() {  
    viewModelScope.launch(Dispatchers.IO +  
        errorHandler) {
```



```
        val restaurants =  
        restInterface.getRestaurants()  
            withContext(Dispatchers.Main) {  
                state.value =  
                restaurants.restoreSelections()  
            }  
    }  
}
```

The `getRestaurants()` method of the `restInterface`:

`RestaurantsApiService` interface is a suspending function. This function should always be run on `Dispatchers.IO` since it executes a heavy I/O operation, that is, the network request.

However, this would mean that whenever we have to call `restInterface.getRestaurants()`, we either have to call this suspending function from a coroutine that has a scope of `Dispatchers.IO` – just as we did previously – or always wrap it in a `withContext(Dispatchers.IO)` block inside the caller coroutine.

Both of these alternatives don't scale well. Imagine that you have to call `restInterface.getRestaurants()` 10 times in the `RestaurantsViewModel` class. You would always have to be careful with setting the dispatcher when calling this function.

Let's address this by creating a separate method where we can specify the correct dispatcher for our suspending function:

1. Inside the `RestaurantsViewModel` class, create a separate suspending method, called `getRemoteRestaurants()`, and wrap the `restInterface.getRestaurants()` call there with a `withContext()` block:

```
private suspend fun getRemoteRestaurants():  
    List<Restaurant> {  
        return withContext(Dispatchers.IO) {  
            restInterface.getRestaurants()  
        }  
    }
```

```
}
```

To the `withContext` method, we've passed the corresponding dispatcher for this suspending function: `Dispatchers.IO`.

This means that whenever this suspending function is called (from a coroutine or another suspending function), the dispatcher will be switched to `Dispatchers.IO` for the `restInterface.getRestaurants()` call's execution.

By doing so, we make sure that whoever is calling `getRemoteRestaurants()` will not have to care about the correct thread dispatcher for the content of this method.

2. In the `getRestaurants()` method of the `ViewModel` component, replace the `restInterface.getRestaurants()` method call with `getRemoteRestaurants()`:

```
private fun getRestaurants() {
    viewModelScope.launch(Dispatchers.IO +
        errorHandler)
    {
        val restaurants = getRemoteRestaurants()
        withContext(Dispatchers.Main) {
            state.value =
                restaurants.restoreSelections()
        }
    }
}
```

3. Since the content of the `getRemoteRestaurants()` method will be called on its appropriate dispatcher, we no longer have to pass `Dispatchers.IO` to the launch block. Remove the `Dispatchers.IO` dispatcher from the coroutine `launch` block:

```
private fun getRestaurants() {
    viewModelScope.launch(errorHandler) {
        val restaurants = getRemoteRestaurants()
```

```
        withContext(Dispatchers.Main) {  
            state.value = restaurants.  
                restoreSelections()  
        }  
    }  
}
```

By default, the launch block will inherit the **CoroutineContext** (and so its defined **Dispatcher** object) from its parent coroutine. In our case, there is no parent coroutine, so the **launch** block will launch a coroutine on the **Dispatchers.Main** thread which was predefined by the **viewModelScope** custom scope.

4. Since the coroutine will now run on the **Dispatchers.Main** thread, we can remove the redundant **withContext(Dispatchers.Main)** block from within the **getRestaurants()** method. The **getRestaurants()** method should now look like this:

```
private fun getRestaurants() {  
    viewModelScope.launch(errorHandler) {  
        val restaurants = getRemoteRestaurants()  
        state.value =  
            restaurants.restoreSelections()  
    }  
}
```

Now, the **getRestaurants()** method where we launched the coroutine is much easier to read and understand. Our suspending function call, for instance, **getRemoteRestaurants()**, is called inside this coroutine on the **Dispatchers.Main** dispatcher. However, at the same time, our suspending function has its own **withContext()** block with its corresponding **Dispatcher** object set:

```
private suspend fun getRemoteRestaurants()  
    : List<Restaurant> {  
    return withContext(Dispatchers.IO) {
```

```
        restInterface.getRestaurants()  
    }  
}
```

This practice allows us to call suspending functions from coroutines with any given **Dispatcher** object, simply because the suspending functions have their own **CoroutineContext** object set with their appropriate **Dispatcher** objects.

At runtime, even though the coroutines are launched on their initial **Dispatcher** object, when our suspending functions are called, the **Dispatcher** object is briefly overridden for every suspending function that is internally wrapped with a **withContext** block.

### NOTE

*For Retrofit interface calls such as `restInterface.getRestaurants()`, we can skip wrapping them in `withContext()` blocks because Retrofit already does this behind the scenes and sets the **Dispatchers.IO** dispatcher for all suspending methods from within its interface.*

Finally, the application should behave the same. However, in terms of good practices, we made sure that the correct **Dispatcher** object is set for every suspending function out of the box, and without us having to manually set it in every coroutine.

Now that we improved the way dispatchers are set within our suspending function and coroutine, it's time to wrap this chapter up.

## Summary

In this chapter, we learned how coroutines allow us to write async code in a much clearer and more concise way.

We understood what coroutines are, how they work, and why they are needed in the first place. We unveiled the core elements of coroutines:

from `suspend` functions to `CoroutineScope` objects, to `CoroutineContext` objects and `Dispatcher` objects.

Then, we replaced the callbacks with coroutines in our Restaurants application and noticed how the code is much easier to understand and less nested. Additionally, we learned how to perform error handling with coroutines and integrated some of the best practices when working with coroutines.

In the next chapter, we will add another Compose-based screen to our Restaurants application and learn how to navigate between screens in Compose with yet another Jetpack library.

## Further reading

While canceling coroutines might seem simple with the help of the associated `Job` objects, it's important to note that any cancelation must be cooperative. More specifically, when coroutines perform suspending work based on conditional statements, you must ensure the coroutine is cooperative with respect to canceling.

You can read about this topic, in more detail, in the official documentation: <https://kotlinlang.org/docs/cancellation-and-timeouts.html#cancellation-is-cooperative>.

[Support](#) | [Sign Out](#)

©2022 O'REILLY MEDIA, INC. [TERMS OF SERVICE](#) | [PRIVACY POLICY](#)