

Chapter 6: Handling Flow Cancelations and Exceptions

In the previous chapter, we focused on Kotlin Flows and learned how we can use them in our Android projects. We learned about creating Kotlin Flows with Flow builders. We then explored Flow operators and how to use them with Kotlin Flows. We then learned about buffering and combining Flows. Finally, we explored **SharedFlow** and **StateFlow**.

Flows can be canceled, and they can fail or encounter exceptions. Developers must be able to handle these properly to prevent application crashes and to inform their users with a dialog or a toast message. We will discuss how to do these tasks in this chapter.


In this chapter, we will start by understanding Flow cancelation. We will learn how to cancel Flows and handle cancelations for our Flows. Then, we will learn how to manage failures and exceptions that can happen in our Flows. We will also learn about retrying and handling Flow completion.

In this chapter, we are going to cover the following main topics:


- Canceling Kotlin Flows
- Retrying tasks with Flow
- Catching exceptions in Flows
- Handling flow completion

By the end of this chapter, you will understand how to cancel flows, and will have learned how to manage cancelations and how to handle exceptions in flows.

Technical requirements

You will need to download and install the latest version of Android Studio. You can find the latest version at <https://developer.android.com/studio> . For an optimal learning experience, a computer with the following specifications is recommended:

- Intel Core i5 or equivalent or higher
- 4 GB RAM minimum
- 4 GB available space

The code examples for this chapter can be found on GitHub at <https://github.com/PacktPublishing/Simplifying-Android-Development-with-Coroutines-and-Flows/tree/main/Chapter06> 

Canceling Kotlin Flows

In this section, we will start by looking at Kotlin Flow cancelations. Like coroutines, Flows can also be canceled manually or automatically.

In [Chapter 3](#), *Handling Coroutine Cancelations and Exceptions*, we learned about canceling coroutines and that coroutine cancellation must be cooperative. As Kotlin Flows are built on top of coroutines, Flow follows the cooperative cancellation of coroutines.

Flows created using the `flow{}` builder are cancellable by default. Each `emit` call to send new values to the Flow also calls `ensureActive` internally. This checks whether the coroutine is still active, and if not, it will throw `CancellationException`.

For example, we can use the `flow{}` builder to create a cancellable Flow, as shown in the following:

```
class MovieViewModel : ViewModel() {  
  
    ...  
}
```

```
fun fetchMovies(): Flow<Movie> = flow {  
  
    movieRepository.fetchMovies().forEach {  
  
        emit(it)  
  
    }  
  
}
```

In the **fetchMovies** function here, we used the **flow** builder to create the Flow of movies returned by **movieRepository.fetchMovies**. This **Flow<Movie>** will be a cancellable Flow by default.

All other Flows, such as ones created using the **asFlow** and **flowOf** Flow builders, are not cancellable by default. We must handle the cancellation ourselves. There is a **cancellable()** operator we can use on a Flow to make it cancelable. This will add an **ensureActive** call on each emission of a new value.

The following example shows how we can make a Flow cancelable using the **cancellable** Flow operator:

```
class MovieViewModel : ViewModel() {  
  
    ...  
  
    fun fetchMovies(): Flow<Movie> {  
  
        return movieRepository.fetchMovies().cancellable()  
  
    }  
  
}
```

In this example, we used the cancelable operator on the Flow returned by `movieRepository.fetchMovies()` to make the resulting Flow cancelable.

In this section, we learned how to cancel Kotlin Flows and how to make sure your Flows can be cancellable. In the next section, we will focus on how to retry your tasks with Kotlin Flows.

Retrying tasks with Flow

In this section, we will explore Kotlin Flow retrying. There are cases when retrying an operation is needed for your application.

When performing long-running tasks, such as a network call, sometimes it is necessary to try the call again. This includes cases such as logging in/out, posting data, or even fetching data. The user may be in an area with a low internet connection, or there may be other factors why the call is failing. With Kotlin Flows, we have the `retry` and `retryWhen` operators that we can use to retry Flows automatically.

The `retry` operator allows you to set a **Long retries** as the maximum number of times the Flow will retry. You can also set a **predicate** condition, a code block that will retry the Flow when it returns **true**. The predicate has a **Throwable** parameter representing the exception that occurred; you can use that to check whether you want to do the retry or not.

The following example shows how we can use the `retry` Flow operator to retry our tasks in our Flow:

```
class MovieViewModel : ViewModel() {  
  
    ...  
  
    fun favoriteMovie(id: Int) =  
  
        movieRepository.favoriteMovie(id)
```

```
        .retry(3) { cause -> cause is IOException }  
    }  
}
```

Here, the Flow from `movieRepository.favoriteMovie(id)` will be retried up to three times when the exception encountered is `IOException`.

If you do not pass a value for the retries, the default of `Long.MAX_VALUE` will be used. `predicate`, when not provided, has a default value of `true`, meaning the Flow will always be retried if `retries` has not yet been reached.

The `retryWhen` operator is similar to the `retry` operator. We need to specify `predicate`, which is the condition and only when `true` will it perform the retry. `predicate` has a `Throwable` parameter representing the exception encountered and a `Long` parameter for the number of attempts (which starts at zero). We can use both to create the condition which, if evaluated to `true`, will retry the Flow. The following code shows an example of using `retryWhen` to retry your tasks in your Flow:

```
class MovieViewModel : ViewModel() {  
    ...  
  
    fun favoriteMovie(id: Int) =  
  
        movieRepository.favoriteMovie(id)  
  
            .retryWhen { cause, attempt -> attempt < 3 &&  
                cause is IOException }  
}
```

In this example, we used `retryWhen` and specified that the retry will be done when the value of `attempt` is less than three and only if the exception is `IOException`.

With the **retryWhen** operator, we can also emit a value to the Flow (with the **emit** function), which we can use to represent the retry attempt or a value. We can then display this value on the screen or process it. The following example shows how we can use **emit** with the **retryWhen** operator:

```
class MovieViewModel : ViewModel() {  
  
    ...  
  
    fun getTopMovieTitle(): Flow<String> {  
  
        return movieRepository.getTopMovieTitle(id)  
  
            .retryWhen { cause, attempt ->  
  
                emit("Fetching title again...")  
  
                attempt < 3 && cause is IOException  
  
            }  
  
        ...  
  
    }  
}
```

Here, the Flow's task will be retried when the number of attempts is less than three and only if the exception is **IOException**. It will then emit a **Fetching title again** string that can be processed by the activity or fragment that listens to the Flow returned by **MovieViewModel.getTopMovieTitle()**.

In this section, you learned about retrying tasks such as network requests with Kotlin Flow. We will explore Kotlin Flow exceptions and how to update our code to catch these exceptions in the next section.

Catching exceptions in Flows

The Flows in your code can encounter **CancellationException** when they are canceled or other exceptions when emitting or collecting values. In this section, we will learn how to handle these Kotlin Flow exceptions.

Exceptions can happen in Flows during the collection of values or when using any operators on a Flow. We can handle exceptions in Flows by enclosing the collection of the Flow in our code with a **try-catch** block. For example, in the following code, the **try-catch** block is used to add exception handling:

```
class MainActivity : AppCompatActivity() {  
  
    ...  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
  
        ...  
  
        lifecycleScope.launch {  
  
            repeatOnLifecycle(Lifecycle.State.STARTED) {  
  
                try {  
  
                    viewModel.fetchMovies().collect { movie ->  
  
                        processMovie(movie)  
  
                    }  
  
                } catch (exception: Exception) {  
  
                    Log.e("Error", exception.message)  
  
                }  
  
            }  
  
        }  
  
    }  
}
```

```

        }

    }

}

}

```

Here, the collection code for the Flow returned by `viewModel.fetchMovies` was wrapped in a **try-catch** block. If an exception was encountered in the Flow, the exception message will be logged with the **Error** tag and **exception.message** as the message.

We can also use the **catch** Flow operator to handle exceptions in our Flow. With the **catch** operator, we can catch the exceptions from the upstream Flow, or the function and operators before the **catch** operator was called.

In the following example, the **catch** operator was used to catch exceptions from the Flow returned by `viewModel.fetchMovies`:

```

class MainActivity : AppCompatActivity() {

    ...

    override fun onCreate(savedInstanceState: Bundle?) {

        ...

        lifecycleScope.launch {

            repeatOnLifecycle(Lifecycle.State.STARTED) {

                viewModel.fetchMovies()

                    .catch { exception ->

                        handleException(exception) }
            }
        }
    }
}

```



```

        .collect { movie -> processMovie(movie) }

    }

}

}

}

```

Here, the **catch** operator was used in the Flow to catch the exceptions. The exception, which is an instance of **Throwable**, was then passed to the **handleException** function that is going to handle the exception.

We can also use the **catch** operator to emit a new value to represent the error or for use as a fallback value instead, such as an empty list. In the following example, a default string value of **No Movie Fetched** will be used when an exception occurs in the Flow that returns the title of the top movie:

```

class MainActivity : AppCompatActivity() {

    ...

    override fun onCreate(savedInstanceState: Bundle?) {

        ...

        lifecycleScope.launch {

            repeatOnLifecycle(Lifecycle.State.STARTED) {

                viewModel.getTopMovieTitle()

                    .catch { emit("No Movie Fetched") }

                    .collect { title -> displayTitle(title) }

            }

        }

    }

}

```

```

    }

}

}

```

In this example, we used the **catch** operator to emit the **No Movie Fetched** string when an exception occurs in getting the top movie title from **ViewModel**. This will be the value that will be used in the **displayTitle()** call.

As the **catch** operator only handles exceptions in the upstream Flow, an exception that happens during the **collect{}** call won't be caught. While you can use the **try-catch** block to handle these exceptions, you can also move the collection code to an **onEach** operator, add the **catch** operator after it, and use **collect()** to start the collection.

The following example shows how your code can look when using an **onEach** operator for the collection of values and the **catch** operator for handling exceptions:

```

class MainActivity : AppCompatActivity() {

    ...

    override fun onCreate(savedInstanceState: Bundle?) {

        ...

        lifecycleScope.launch {

            repeatOnLifecycle(Lifecycle.State.STARTED) {

                viewModel.fetchMovies()

                    .onEach { movie -> processMovie(movie) }

```

```
        .catch { exception ->

            handleError(exception) }

        .collect()

    }

}

}
```

Here, the **collect()** function without parameters was used, and the **onEach** operator will process each movie from the Flow.

In this section, we learned how to catch exceptions in Flows. In the following section, we will focus on Kotlin Flow completion.

Handling Flow completion

In this section, we will explore how to handle Flow completion. We can add code to perform additional tasks after our Flows have completed.

When the Flow encounters an exception, it will be canceled and complete the Flow. A Flow is also completed when the last element of the Flow has been emitted.

To add a listener in your Flow when it has completed, you can use the **onCompletion** operator and add the code block that will be run when the Flow completes. A common usage of **onCompletion** is hiding the **ProgressBar** in your UI when the Flow has completed, as shown in the following code:

```
class MainActivity : AppCompatActivity() {
```

```
...

override fun onCreate(savedInstanceState: Bundle?) {

    ...

    lifecycleScope.launch {

        repeatOnLifecycle(Lifecycle.State.STARTED) {

            viewModel.fetchMovies()

                .onStart { progressBar.isVisible = true }

                .onEach { movie -> processMovie(movie) }

                .onCompletion { progressBar.isVisible =

                    false }

                .catch { exception ->

                    handleError(exception) }

                .collect()

        }

    }

}
```

In this example, we have added the **onCompletion** operator to hide **progressBar** when the Flow has completed. We have also used **onStart** to display **progressBar**.

The **onStart** operator is the opposite of **onCompletion**. It will be called before the Flow starts emitting values. In the previous example, **onStart** was

used so that before the Flow starts, **progressBar** will be displayed on the screen.

Within the code block you add in **onStart** and **onCompletion** (if the Flow completed successfully and without exception), you can also emit values, such as an initial and final value. In the following example, an **onStart** operator is used to emit an initial value to be displayed on the screen:

```
class MainActivity : AppCompatActivity() {  
  
    ...  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
  
        ...  
  
        lifecycleScope.launch {  
  
            repeatOnLifecycle(Lifecycle.State.STARTED) {  
  
                viewModel.getTopMovieTitle()  
  
                    .onStart { emit("Loading...") }  
  
                    .catch { emit("No Movie Fetched") }  
  
                    .collect { title -> displayTitle(title) }  
  
            }  
  
        }  
  
    }  
  
}
```

Here, **onStart** is used to listen to when the Flow starts. When the Flow starts, it will emit a **Loading...** string as the initial value of the Flow. This will then be the first item that will be displayed on the screen.

The **onCompletion** code block also has a nullable **Throwable** that corresponds to the exception thrown by the Flow. It will be null if the Flow has completed successfully. However, unlike **catch**, the exception itself will not be handled, so you still need to use **catch** or **try-catch** to handle this exception.

The following example shows how we can use this nullable **Throwable** in our Flow's **onCompletion** call:

```
class MainActivity : AppCompatActivity() {  
  
    ...  
  
    override fun onCreate(savedInstanceState: Bundle?) {  
  
        ...  
  
        lifecycleScope.launch {  
  
            repeatOnLifecycle(Lifecycle.State.STARTED) {  
  
                viewModel.getTopMovieTitle()  
  
                    .onCompletion { cause ->  
  
                        progressBar.isVisible = false  
  
                        if (cause != null) displayError(cause)  
  
                    }  
  
                    .catch { emit("No Movie Fetched") }  
  
                    .collect { title -> displayTitle(title) }  
  
                }  
  
            }  
  
        }  
    }  
}
```

```
}  
  
}
```

In this example, we checked the cause in the **onCompletion** block, and if it's not null (which means an exception was encountered), **displayError** will be called and the cause passed to it.

In this section, we learned about **onStart** and **onCompletion** to handle when Flows start and when they are completed.

Let's try what you have learned by adding code to handle exceptions that can occur in Flows in an Android project.

Exercise 6.01 – Handling Flow exception in an Android app

In this exercise, you will be continuing with the movie app you worked on in *Exercise 5.01 – Using Kotlin Flow in an Android app*. This application displays the movies that are playing now in movie theaters. You will be updating the project to handle Flow cancellations and exceptions by following these steps:

1. In Android Studio, open the movie app you worked on in *Exercise 5.01 – Using Kotlin Flow in an Android app*.
2. Go to the **MovieViewModel** class. In the **fetchMovies** function, remove the line that sets the value of **_loading** to **true**. Your function will look like the following:

```
fun fetchMovies() {  
    viewModelScope.launch (dispatcher) {  
        MovieRepository.fetchMoviesFlow()  
            .collect {  
                _movies.value = it  
                _loading.value = false  
            }  
    }  
}
```

```

        }
    }
}

```

You removed the code that sets **loading** to **true** (and displays **ProgressBar** on the screen). It will be replaced in the next step with an **onStart** Flow operator.

3. Add an **onStart** operator before the **collect** call, which will set the value of **_loading** to **true** when the Flow starts, as shown in the following:

```

fun fetchMovies() {
    viewModelScope.launch (dispatcher) {
        MovieRepository.fetchMoviesFlow()
            .onStart { _loading.value = true }
            .collect {
                _movies.value = it
                _loading.value = false
            }
    }
}

```

The **onStart** operator will set the value of **_loading** to **true** and display **ProgressBar** on the screen when the Flow starts.

4. Next, remove the line that sets the value of **_loading** to **false** in the code block inside the **collect** call. Your function will look like the following:

```

fun fetchMovies() {
    viewModelScope.launch (dispatcher) {
        MovieRepository.fetchMoviesFlow()
            .onStart { _loading.value = true }
            .collect {
                _movies.value = it
            }
    }
}

```



```
    }  
}
```

You removed the code that sets the value of `_loading` to **false** and hides **ProgressBar** on the screen when the Flow is collected.

5. Add an **onCompletion** operator before the **collect** call, which will set the value of `_loading` to **false** when the Flow has completed, as shown in the following:

```
fun fetchMovies() {  
    viewModelScope.launch (dispatcher) {  
        MovieRepository.fetchMoviesFlow()  
            .onStart { _loading.value = true }  
            .onCompletion { _loading.value = false  
  
    }  
  
        .collect {  
            _movies.value = it  
        }  
    }  
}
```

The **onCompletion** Flow operator will set the value of `_loading` to **false**. This will then hide, upon completion of the Flow, **ProgressBar**, which is displayed on the screen while the movies are being fetched.

6. Add a **catch** operator before the **collect** function to handle the case when the Flow has encountered an exception:

```
fun fetchMovies() {  
    viewModelScope.launch (dispatcher) {  
        MovieRepository.fetchMoviesFlow()  
            .onStart { _loading.value = true }  
            .onCompletion { _loading.value = false  
  
    }  
  
        .catch {
```

```
        _error.value = "An exception  
occurred:  
        ${it.message}"  
    }  
    .collect {  
        _movies.value = it  
    }  
}  
}
```

This will set a string containing **An exception occurred:** and the exception message as the value of the `_error` LiveData. This `_error` LiveData will display an error message in **MainActivity**.

7. On your device or emulator, turn off the Wi-Fi and mobile data. Then, run the app. This will cause an error in fetching the movies, as there is no internet connection. The app will display a **SnackBar** message, as shown in the following screenshot:

Figure 6.1 – The error message displayed in the movie app

8. Close the application and turn on the Wi-Fi and/or mobile data on your device or emulator. Run the application again. The app should show **ProgressBar**, display a list of movies (with the movie title and

poster) on the screen, and hide **ProgressBar**, as shown in the following screenshot:

Figure 6.2 – The movie app with the list of movies

In this exercise, you have updated the application so that it can handle exceptions in the Flow instead of crashing.

Summary

This chapter focused on Kotlin Flow cancelations. You learned that Flows follow the cooperative cancellation of coroutines. The `flow{}` builder and `StateFlow` and `SharedFlow` implementations are cancellable by default. You can use the `cancellable` operator to make other Flows cancellable.

We then learned about retrying tasks with Kotlin Flow. You can use the `retry` and `retryWhen` functions to retry the Flow based on the number of attempts and the exception encountered by the Flow.

Then, we learned about handling exceptions that can happen during the emission or collection of data in a Flow. You can use the `try-catch` block or the `catch` Flow operator to handle Flow exceptions.

We learned how to handle Flow completion. With the `onStart` and `onCompletion` operators, you can listen and run code when Flows start and when they have finished. You can also emit values with the `onStart` and `onCompletion` code blocks, such as when you want to set an initial and final value for the Flow.

Finally, we worked on an exercise to update our Android project and handle the exceptions that can be encountered in a Flow. We used the `catch` Flow operator to handle exceptions in the project.

In the next chapter, we will dive into creating and running tests for the Kotlin Flows in our Android projects.

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

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