Chapter 5: Using Kotlin Flows

In the previous three chapters, we dove into Kotlin coroutines and learned how we can use them for asynchronous programming in Android. We learned about coroutine builders, scopes, dispatchers, contexts, and jobs. We then learned how to handle coroutine cancelations, timeouts, and exceptions. We also learned how to create tests for coroutines in your code.

In the next three chapters, we will focus on Kotlin Flow, a new asynchronous stream library built on top of Kotlin coroutines. A flow can emit multiple values over a length of time instead of just a single value. You can use Flows for streams of data, such as real-time location, sensor readings, and live database values.

In this chapter, we will explore Kotlin Flows. We will start by building Kotlin Flows. Then, we will look into the various operators you can use for transforming, combining, buffering, and doing more with Flows. Finally, we will learn about StateFlows and SharedFlows.

This chapter covers the following main topics:

- Using Flows in Android
- Creating Flows with Flow builders
- Using operators with Flows
- Buffering and combining Flows
- Exploring StateFlow and SharedFlow

By the end of this chapter, you will have a deeper understanding of using Kotlin Flows. You will be able to use Flows for various cases in your Android apps. You will also learn about flow builders, operators, combining flows, StateFlow, and SharedFlow.

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/Chapter05 ↗️.

Using Flows in Android

In this section, we will start by using flows in Android for asynchronous programming. Flows are ideal for the parts of your application that involve live data updates.

A Flow of data is represented by the **kotlinx.coroutines.flow.Flow** interface. Flows emit multiple values of the same type one at a time. For example, `Flow<String>` is a flow that emits string values.

Android Jetpack libraries such as Room, Paging, DataStore, WorkManager, and Jetpack Compose include built-in support for Flow.

The Room database library added support for Flows, starting with version 2.2. This allows you to be notified of changes in the database values by using Flows.

If your Android application uses a **Data Access Object** (**DAO**) to display a list of movies, your project can have a DAO such as the following:

```
@Dao

interface MovieDao {

    @Query("SELECT * FROM movies")

    fun getMovies(): List<Movie>

    ...

}
```

By calling the **getMovies** function from **MovieDao**, you can get the list of movies from the database.

The preceding code will only fetch the list of movies once, after calling **getMovies**. You may want your application to automatically update the list of movies whenever a movie in the database has been added, removed, or updated. You can do that by using Room-KTX and changing your **MovieDao** to use Flow for **getMovies**:

```
@Dao

interface MovieDao {

    @Query("SELECT * FROM movies")

    fun getMovies(): Flow<List<Movie>>

    ...

}
```

With this code, every time the **movies** table has a change, **getMovies** will emit a new list containing the list of movies from the database. Your application can then use that to automatically update the movies displayed in your List or **RecyclerView**.

If you are using **LiveData** and want to convert **LiveData** to **Flow**, or **Flow** to **LiveData**, you can use the LiveData KTX.

To convert **LiveData** to **Flow**, you can use the **LiveData.asFlow()** extension function. With the **Flow.asLiveData()** extension function to convert **Flow** to **LiveData**. You can add LiveData KTX to your project by including the following to your **app/build.gradle** dependencies:

```
dependencies {
    ...

    implementation ‘androidx.lifecycle:lifecycle-livedata-
      ktx:2.2.0'
}
```

This adds the LiveData KTX to your project, allowing you to use the **as-Flow()** and **asLiveData()** extension functions to convert **LiveData** to **Flow** and **Flow** to **LiveData**.

Third-party Android libraries now also support Flows; some functions can return Flow objects. If you are using RxJava 3 in your project, you can use the **kotlinx-coroutines-rx3** library to convert **Flow** to **Flowable** or **Observable** and vice versa.

A flow will only start emitting values when you call the **collect** function. The **collect** function is a suspending function, so you should call it from a coroutine or another suspending function.

In the following example, the **collect()** function was called from the coroutine created using the **launch** coroutine builder from **lifecycleScope**:

```
class MainActivity : AppCompatActivity() {

    ...

    override fun onCreate(savedInstanceState: Bundle?) {

        ...

        lifecycleScope.launch {

            viewModel.fetchMovies().collect { movie ->

                Log.d("movies", "${movie.title}")

            }

        }

    }

}

class MovieViewModel : ViewModel() {

    ...

    fun fetchMovies(): Flow<Movie> {

        ...

    }

}
```

In this example, the **`collect{}`** function was called on **`Flow<Movie>`** and returned by calling **`viewModel.fetchMovies()`**. This will cause the Flow to start emitting values, and you can then process each value.

The collection of the flow occurs in **`CoroutineContext`** of the parent coroutine. In the previous example, the coroutine context is from **`viewMod-`**

elScope.

To change **CoroutineContext** where the Flow is run, you can use the **flowOn()** function. If you want to change **Dispatcher** on the Flow in the previous example to **Dispatchers.IO)**, you can use **flowOn(Dispatchers.IO)**, as shown in the following example:

```
class MainActivity : AppCompatActivity() {

    ...

    override fun onCreate(savedInstanceState: Bundle?) {

        ...

        lifecycleScope.launch {

            viewModel.fetchMovies()

                .flowOn(Dispatchers.IO)

                .collect { movie ->

                    Log.d("movies", "${movie.title}")

            }

        }

    }

}
```

Here, before collecting the Flow, the dispatcher where the Flow is run was changed to **Dispatchers.IO** by calling **flowOn** with **Dispatchers.IO**.

When you call **flowOn**, it will only change the preceding functions or operators and not the ones after you called it. In the following example, a **map** operator was called after the **flowOn** call to change the dispatcher, so its context won't be changed:

```
class MainActivity : AppCompatActivity() {

    ...

    override fun onCreate(savedInstanceState: Bundle?) {

        ...

        lifecycleScope.launch {

            viewModel.fetchMovies()

                .flowOn(Dispatchers.IO)

                .map { ... }

                .collect { movie ->

                    Log.d("movies", "${movie.title}")

            }

        }

    }

}
```

In this example, **flowOn** will only change the context of the ones preceding the call, so the **map** call will not be changed. It will still use the original context (which is the one from **lifecycleScope**).

In Android, you can collect Flow in the Fragment or Activity classes to display the data in the UI. If the UI goes to the background, your Flow will keep on collecting the data. Your app must not continue collecting the

Flow and updating the screen to prevent memory leaks and avoid wasting resources.

To safely collect flows in the Android UI layer, you would need to handle the lifecycle changes yourself. You can also use **`Lifecycle.repeatOnLifecycle`** and **`Flow.flowWithLifecycle`**, which are available in the **lifecycle-runtime-ktx** library, starting with version 2.4.0. To add it to your project, you can add the following to your **`app/build.gradle`** dependencies:

```
dependencies {

    ...

    implementation ‘androidx.lifecycle:lifecycle-runtime-

      ktx:2.4.1

}
```

This adds the **lifecycle-runtime-ktx** library to your project, allowing you to use **`Lifecycle.repeatOnLifecycle`** and **`Flow.flowWithLifecycle`**.

`Lifecycle.repeatOnLifecycle(state, block)` suspends the parent coroutine until the lifecycle is destroyed and executes the suspending **`block`** of code when the lifecycle is at least in **`state`** you set. When the lifecycle moves out of the state, **`repeatOnLifecycle`** will stop the Flow and restart it when the lifecycle moves back to the said state.

If you used **Lifecycle.State.STARTED** for the state, your **`repeatOnLifecycle`** will start collecting the Flow whenever the lifecycle is started. It will stop when the lifecycle is stopped, when the **`onStop()`** of the lifecycle is called.

When you use **Lifecycle.State.RESUMED** for the state, your **`repeatOnLifecycle`** will start collecting the Flow every time the lifecycle is resumed and will stop when the lifecycle is paused or when **`onPause()`** is called.

It is recommended to call **`Lifecycle.repeatOnLifecycle`** on the activity's **`onCreate`** or on the fragment's **`onViewCreated`** functions.

The following shows how you can use **`Lifecycle.repeatOnLifecycle`** in your Android project:

```
class MainActivity : AppCompatActivity() {

    ...

    override fun onCreate(savedInstanceState: Bundle?) {

        ...

        lifecycleScope.launch {

            repeatOnLifecycle(Lifecycle.State.STARTED) {

                viewModel.fetchMovies()

                    .collect { movie ->

                        Log.d("movies", "${movie.title}")

                    }

            }

        }

    }

}
```

Here, we used **`repeatOnLifecycle`** with **Lifecycle.State.STARTED** to start collecting the Flow of movies when the lifecycle is started and stop when the lifecycle is stopped.

You can use **`Lifecycle.repeatOnLifecycle`** to collect more than one Flow. To do so, you must collect them in parallel in different coroutines:

```
class MainActivity : AppCompatActivity() {
    ...
  override fun onCreate(savedInstanceState: Bundle?) {
      ...
      lifecycleScope.launch {
          repeatOnLifecycle(Lifecycle.State.STARTED) {
              launch {
                 viewModel.fetchMovies().collect { movie ->
                      Log.d("movies", "${movie.title}")
                    }
                }
              launch {
                  viewModel.fetchTVShows.collect { show ->
                      Log.d("tv shows", "${show.title}")
                    }
                }
            }
        }
    }
}
```

Here, there are two Flows: one to collect the movies and the other to collect the TV shows. The collections of the Flow are started from separate **`launch`** coroutine builders.

If you only have one Flow to collect, you can also use **`Flow.flowWithLifecycle`**. This emits values from the upstream Flow (the Flow and operators preceding the call) when the lifecycle is at least in **Lifecycle.State.STARTED** or the state you set. It uses **`Lifecycle.repeatOnLifecycle`** internally. You can use **`Flow.flowWithLifecycle`** as shown in the following code:

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        lifecycleScope.launch {
            viewModel.fetchMovies()
                .flowWithLifecycle(lifecycle,
                  Lifecycle.State.STARTED)
                .collect { movie ->
                    Log.d("movies", "${movie.title}")
            }
        }
    }
}
```

In this example, you used **`flowWithLifecycle`** with **Lifecycle.State.STARTED** to start collecting the Flow of movies when the lifecycle is started and stop if the lifecycle is stopped.

In this section, you have learned about using Kotlin Flows in your Android app. You can use Flow in Android Jetpack libraries such as Room and even in third-party libraries. To safely collect flows in the UI layer and prevent memory leaks and avoid wasting resources, you can use **`Lifecycle.repeatOnLifecycle`** and **`Flow.flowWithLifecycle`**.

In the next section, we will be looking into the different Flow builders you can use to create Flows for your application.

Creating Flows with Flow builders

In this section, we will start by looking at creating Flows. To create a Flow, you can use a Flow builder.

The Kotlin Flow API has flow builders that you can use to create Flows. The following are the Kotlin Flow builders you can use:

- **`flow {}`**
- **`flowOf()`**
- **`asFlow()`**

The **`flow`** builder function creates a new Flow from a suspendable lambda block. Inside the block, you can send values using the **`emit`** function. For example, this **`fetchMovieTitles`** function of **`MovieViewModel`** returns **`Flow<String>`**:

```
class MovieViewModel : ViewModel() {

    ...
    fun fetchMovieTitles(): Flow<String> = flow {
        val movies = fetchMoviesFromNetwork()
        movies.forEach { movie ->
            emit(movie.title)
        }
    }
    private fun fetchMoviesFromNetwork(): List<Movie> {
         …
    }
}
```

In this example, **fetchMovieTitles** created a Flow with the movie titles. It iterated over the list of movies from **fetchMoviesFromNetwork** and, for each movie, emitted the movie's title with the **emit** function.

With the **flowOf** function, you can create a Flow that produces the specified value or **vararg** values. In the following example, the **flowOf** function is used to create a Flow of the titles of the top three movies:

```
class MovieViewModel : ViewModel() {
    ...
    fun fetchTop3Titles(): Flow<List<Movie>> {
        val movies = fetchMoviesFromNetwork().sortedBy {
            it.popularity }
        return flowOf(movies[0].title,
            movies[1].title,
            movies[2].title)
    }
    private fun fetchMoviesFromNetwork(): List<Movie> {
        …
    }
}
```

Here, **fetchTop3Titles** uses **flowOf** to create a Flow containing the titles of the first three movies.

The **asFlow()** extension function allows you to convert a type into a Flow. You can use this on sequences, arrays, ranges, collections, and functional types. For example, this **MovieViewModel** has **fetchMovieIds** that returns **Flow<Int>**, containing the movie IDs:

```
class MovieViewModel : ViewModel() {
    ...
    private fun fetchMovieIds(): Flow<Int> {
        val movies: List<Movie> = fetchMoviesFromNetwork()
        return movies.map { it.id }.asFlow()
    }
    private fun fetchMoviesFromNetwork(): List<Movie> {
        …
    }
    }
```

In this example, we used a **`map`** function on the list of movies to create a list of the movie IDs. The list of movie IDs was then converted to **`Flow<String>`** by using the **`asFlow()`** extension function on it.

In this section, we learned how you can create Flows with Flow Builders. In the next section, we will check out the various Kotlin Flow operators you can use to transform, combine, and do more with Flows.

Using operators with Flows

In this section, we will focus on the various Flow operators. Kotlin Flow has built-in operators that you can use with Flows. We can collect flows with terminal operators and transform Flows with Intermediate operators.

Collecting Flows with terminal operators

In this section, we will explore the terminal operators you can use on Flows to start the collection of a Flow. The **`collect`** function we used in the previous examples is the most used terminal operator. However, there are other built-in terminal Flow operators.

The following are the built-in terminal Flow operators you can use to start the collection of the Flow:

- **`toList`**: Collects the Flow and converts it into a list
- **`toSet`**: Collects the Flow and converts it into a set
- **`toCollection`**: Collects the Flow and converts it into a collection
- **`count`**: Returns the number of elements in the Flow
- **`first`**: Returns the Flow's first element or throws a **NoSuchElementException** if the Flow was empty

- **`firstOrNull`**: Returns the Flow's first element or null if the Flow was empty
- **`last`**: Returns the Flow's last element or throws a **NoSuchElementException** if the Flow was empty
- **`lastOrNull`**: Returns the Flow's last element or null if the Flow was empty
- **`single`**: Returns the single element emitted or throws an exception if the Flow was empty or had more than one value
- **`singleOrNull`**: Returns the single element emitted or null if the Flow was empty or had more than one value
- **`reduce`**: Applies a function to each item emitted, starting from the first element, and returns the accumulated result
- **`fold`**: Applies a function to each item emitted, starting from the initial value set, and returns the accumulated result

These terminal Flow operators work like the Kotlin collection functions with the same name in the standard Kotlin library.

In the following example, the **`firstOrNull`** terminal operator is used instead of the **`collect`** operator to collect the Flow from **`ViewModel`**:

```
class MainActivity : AppCompatActivity() {

    ...

    override fun onCreate(savedInstanceState: Bundle?) {

        ...

        lifecycleScope.launch {

            repeatOnLifecycle(Lifecycle.State.STARTED) {

                val topMovie =

                  viewModel.fetchMovies().firstOrNull()
```

```
                    displayMovie(topMovie)

                }

            }

        }

    }
```

Here, `firstOrNull` was used on the Flow to get the first item (or null if the Flow was empty), which represents the top movie. It will then be displayed on the screen.

In this section, you learned about the Flow terminal operators you can use to start collecting from a Flow. In the next section, we will learn how to transform Flows with Intermediate operators.

Transforming Flows with Intermediate operators

In this section, we will focus on Intermediate flow operators that you can use to transform Flows. With Intermediate operators, you can return a new Flow based on the original one.

Intermediate operators allow you to modify a Flow and return a new one. You can chain various operators, and they will be applied sequentially.

You can transform the Flow by applying operators on them, as you can do with Kotlin collections. The following Intermediate operators work similarly to the Kotlin collection functions with the same name:

- `filter`: Returns a Flow that selects only the values from the Flow that meet the condition you passed
- `filterNot`: Returns a Flow that selects only the values from the Flow that do not meet the condition you passed

- **filterNotNull**: Returns a Flow that only includes values from the original Flow that are not null
- **filterIsInstance**: Returns a Flow that only includes values from the Flow that are instances of the type you specified
- **map**: Returns a Flow that includes values from the Flow transformed with the operation you specified
- **mapNotNull**: Like **map** (transforms the Flow using the operation specified) but only includes values that are not null
- **withIndex**: Returns a Flow that converts each value to an **IndexedValue** containing the index of the value and the value itself
- **onEach**: Returns a Flow that performs the specified action on each value before they are emitted
- **runningReduce**: Returns a Flow containing the accumulated values resulting from running the operation specified sequentially, starting with the first element
- **runningFold**: Returns a Flow containing accumulated values resulting from running the operation specified sequentially, starting with the initial value set
- **scan**: Like the **runningFold** operator

There is also a **transform** operator that you can use to apply custom or complex operations. With the **transform** operator, you can emit values into the new Flow by calling the **emit** function with the value to send.

For example, this **MovieViewModel** has a **fetchTopMovieTitles** function that uses **transform** to return a Flow with the top movies:

```
class MovieViewModel : ViewModel() {

    ...

    fun fetchTopMovies(): Flow<Movie> {

        return fetchMoviesFlow()

            .transform {
                    if (it.popularity > 0.5f) emit(it)
                }
            }
        }
```

In this example, the **`transform`** operator was used in the Flow of movies to return a new Flow. The **`transform`** operator was used to emit only the list of movies whose popularity is higher than **`0.5`**, which means a popularity of more than 50%.

There are also size-limiting operators that you can use with Flow. The following are some of these operators:

- **`drop(x)`**: Returns a Flow that ignores the first *x* elements
- **`dropWhile`**: Returns a Flow that ignores the first elements that meet the condition specified
- **`take(x)`**: Returns a Flow containing the first *x* elements of the Flow
- **`takeWhile`**: Returns a Flow that includes the first elements that meet the condition specified

These size-limiting operators also function similarly to the Kotlin collection functions with the same name.

In this section, we learned about Intermediate flow operators. Intermediate operators transform a Flow into a new Flow. In the next section, we will learn how to buffer and combine Kotlin Flows.

Buffering and combining flows

In this section, we will learn about buffering and combining Kotlin Flows. You can buffer and combine Flows with Flow operators. Buffering allows Flow with long-running tasks to run independently and avoid race conditions. Combining allows you to join different sources of Flows before processing or displaying them on the screen.

Buffering Kotlin Flows

In this section, we will learn about buffering Kotlin Flows. Buffering allows you to run data emission in parallel to the collection.

Emitting and collecting data with Flow run sequentially. When a new value is emitted, it will be collected. Emission of a new value can only happen once the previous data has been collected. If the emission or the collection of data from the Flow takes a while to complete, the whole process will take a longer time.

With buffering, you can make a Flow's emission and collection of data run in parallel. There are three operators you can use to buffer Flows:

- **`buffer`**
- **`conflate`**
- **`collectLatest`**

`buffer()` allows the Flow to emit values while the data is still being collected. The emission and collection of data are run in separate coroutines, so it runs in parallel. The following is an example of how to use **`buffer`** with Flows:

```
class MainActivity : AppCompatActivity() {

    ...

    override fun onCreate(savedInstanceState: Bundle?) {

        ...

        lifecycleScope.launch {
                repeatOnLifecycle(Lifecycle.State.STARTED) {
                    viewModel.fetchMovies()
                        .buffer()
                        .collect { movie ->
                            processMovie(movie)
                        }
                }
            }
        }
    }
```

Here, the **`buffer`** operator was added before calling **`collect`**. If the **`processMovie(movie)`** function in the collection takes longer, the Flow will emit and buffer the values before they are collected and processed.

`conflate()` is similar to the **`buffer()`** operator, except with **`conflate`**, the collector will only process the latest value emitted after the previous value has been processed. It will ignore the other values previously emitted. Here is an example of using **`conflate`** in a Flow:

```
class MainActivity : AppCompatActivity() {
    ...
    override fun onCreate(savedInstanceState: Bundle?) {
        ...
        lifecycleScope.launch {
                repeatOnLifecycle(Lifecycle.State.STARTED) {
                    viewModel.getTopMovie()
                        .conflate()
                        .collect { movie ->
                            processMovie(movie)
                        }
                }
            }
        }
    }
```

In this example, adding the **`conflate`** operator will allow us to only process the latest value from the Flow and call **`processMovie`** with that value.

`collectLatest(action)` is a terminal operator that will collect the Flow the same way as **`collect`**, but whenever a new value is emitted, it will restart the action and use this new value. Here is an example of using **`collect-Latest`** in a Flow:

```
    class MainActivity : AppCompatActivity() {
        ...
        override fun onCreate(savedInstanceState: Bundle?) {
            ...
            lifecycleScope.launch {
                repeatOnLifecycle(Lifecycle.State.STARTED) {
                    viewModel.getTopMovie()
                        .collectLatest { movie ->
                            displayMovie(movie)
                        }
                }
            }
        }
    }
```

Here, **`collectLatest`** was used instead of the **`collect`** terminal operator to collect the flow from **`viewModel.getTopMovie()`**. Whenever a new value is emitted by this Flow, it will restart and call **`displayMovie`** with the new value.

In this section, you learned how to buffer Kotlin Flows with **`buffer`**, **`conflate`**, and **`collectLatest`**. In the next section, you will learn about combining multiple Flows into a single Flow.

Combining Flows

In this section, we will learn how we can combine Flows. The Kotlin Flow API has available operators that you can use to combine multiple flows.

If you have multiple flows and you want to combine them into one, you can use the following Flow operators:

- **`zip`**
- **`merge`**
- **`combine`**

`merge` is a top-level function that combines the elements from multiple Flows of the same type into one. You can pass a **`vararg`** number of Flows to combine. This is useful when you have two or more sources of data that you want to merge first before collecting.

In the following example, there are two Flows from **`viewModel.fetchMoviesFromDb`** and **`viewModel.fetchMoviesFromNetwork`** combined using **`merge`**:

```
class MainActivity : AppCompatActivity() {

    ...

    override fun onCreate(savedInstanceState: Bundle?) {

        ...

        lifecycleScope.launch {

            repeatOnLifecycle(Lifecycle.State.STARTED) {

                merge(viewModel.fetchMoviesFromDb(),

                  viewModel.fetchMoviesFromNetwork())

                    .collect { movie ->

                        processMovie(movie)

                    }

            }

        }

    }

}
```

In this example, **`merge`** was used to combine the Flows from **`viewModel.fetchMoviesFromDb`** and **`viewModel.fetchMoviesFromNetwork`** before they are collected.

The **`zip`** operator pairs data from the first Flow to the second Flow into a new value using the function you specified. If one Flow has fewer values than the other, **`zip`** will end when the values of this Flow have all been processed.

The following shows how you can use the **`zip`** operator to combine two Flows, **`userFlow`** and **`taskFlow`**:

```
class MainActivity : AppCompatActivity() {

    ...

    override fun onCreate(savedInstanceState: Bundle?) {

        ...

        lifecycleScope.launch {

            repeatOnLifecycle(Lifecycle.State.STARTED) {

                val userFlow = viewModel.getUsers()

                val taskFlow = viewModel.getTasks()

                userFlow.zip(taskFlow) { user, task ->

                    AssignedTask(user, task)

                }.collect { assignedTask ->

                    displayAssignedTask(assignedTask)

                }
                }

            }

        }

    }
```

In this example, you used **`zip`** to pair each value of **`userFlow`** to **`taskFlow`** and return a Flow of **`AssignedTask`** using the **`user`** and **`task`** values. This new Flow will be collected and then displayed with the **`displayAssigned-Task`** function.

`combine` pairs data from the first flow to the second flow like **`zip`** but uses the most recent value emitted by each flow. It will continue to run as long as a Flow emits a value. There is also a top-level **`combine`** function that you can use for multiple flows.

The following example shows how you can use the **`combine`** operator to join two Flows in your application:

```
class MainActivity : AppCompatActivity() {

    ...

    override fun onCreate(savedInstanceState: Bundle?) {

        ...

        lifecycleScope.launch {

            repeatOnLifecycle(Lifecycle.State.STARTED) {

                val yourMesssage =

                  viewModel.getLastMessageSent()

                val friendMessage =
                  viewModel.getLastMessageReceived()
                userFlow.combine(taskFlow) { yourMesssage,
                  friendMessage ->
                    Conversation(yourMessage,
                      friendMessage)
                }.collect { conversation ->
                    displayConversation(conversation)
                }
            }
        }
    }
}
```

Here, you have two Flows, **`yourMessage`** and **`friendMessage`**. The **`combine`** function pairs the most recent value of **`yourMessage`** and **`friendMessage`** to create a **`Conversation`** object. Whenever a new value is emitted by either Flow, **`combine`** will pair the latest values and add that to the resulting Flow for collection.

In this section, we have explored how to combine Flows. In the next section, we will focus on **`StateFlow`** and **`SharedFlow`** and how we can use them in your Android applications.

Exploring StateFlow and SharedFlow

In this section, we will dive into **StateFlow** and **SharedFlow**. **SharedFlow** and **StateFlow** are Flows that are hot streams, unlike a normal Kotlin Flow, which are cold streams by default.

A Flow is a cold stream of data. Flows only emit values when the values are collected. With **SharedFlow** and **StateFlow** hot streams, you can run and emit values the moment they are called and even when they have no listeners. **SharedFlow** and **StateFlow** are Flows, so you can also use operators on them.

A **SharedFlow** allows you to emit values to multiple listeners. **SharedFlow** can be used for one-time events. The tasks that will be done by the **SharedFlow** will only be run once and will be shared by the listeners.

You can use **MutableSharedFlow** and then use the **emit** function to send values to all the collectors.

In the following example, **SharedFlow** is used in **MovieViewModel** for the list of movies fetched:

```
class MovieViewModel : ViewModel() {

    private val _message = MutableSharedFlow<String>()

    val movies: SharedFlow<String> =

      _message.asSharedFlow()

    ...

    fun onError(): Flow<List<Movie>> {

        ...

        _message.emit("An error was encountered")

    }
}
```

In this example, we used **SharedFlow** for the message. We used the **emit** function to send the error message to the Flow's listeners.

StateFlow is **SharedFlow**, but it only emits the latest value to its listeners. **StateFlow** is initialized with a value (an initial state) and keeps this state. You can change the value of **StateFlow** using the mutable version of **StateFlow**, **MutableStateFlow**. Updating the value sends the new value to the Flow.

In Android, **StateFlow** can be an alternative to **LiveData**. You can use **StateFlow** for **ViewModel**, and your activity or fragment can then collect the value. For example, in the following **ViewModel**, **StateFlow** is used for the list of movies:

```
class MovieViewModel : ViewModel() {

    private val _movies =

      MutableStateFlow(emptyList<Movie>())

    val movies: StateFlow<List<Movie>> = _movies

    ...

    fun fetchMovies(): Flow<List<Movie>> {

        ...

        _movies.value = movieRepository.fetchMovies()

    }

}
```

In the preceding code, the list of movies fetched from the repository will be set to **MutableStateFlow** of **_movies**, which will also change **StateFlow** of

movies. You can then collect **StateFlow** of **movies** in an activity or fragment, as shown in the following:

```
class MainActivity : AppCompatActivity() {

    ...

    override fun onCreate(savedInstanceState: Bundle?) {

        ...

        lifecycleScope.launch {

            repeatOnLifecycle(Lifecycle.State.STARTED) {

                viewModel.movies.collect { movies ->

                    displayMovies(movies)

                }

            }

        }

    }

}
```

Here, **StateFlow** of **viewModel.movies** will be collected, and then the list of movies will be displayed on the screen with the **displayMovies** function.

In this section, we have learned about **StateFlow** and **SharedFlow** and how we can use them in our Android projects.

Let's try what we have learned so far by adding Kotlin Flow to an Android project.

Exercise 5.01 – Using Kotlin Flow in an Android app

For this exercise, you will be continuing the movie app you worked on in *Exercise 4.01 – Adding tests to coroutines in an Android app*. This application displays the movies that are currently playing in cinemas. You will be adding Kotlin Flow to the project by following these steps:

1. Open the movie app you worked on in *Exercise 4.01 – Adding tests to coroutines in an Android app* in Android Studio.
2. Go to the **`MovieRepository`** class and add a new **`fetchMoviesFlow()`** function that uses a **`flow`** builder to return a Flow and emits the list of movies from **`MovieService`**, as shown in the following snippet:

```
fun fetchMoviesFlow(): Flow<List<Movie>> {
    return flow {
        emit(movieService.getMovies(apiKey).results
)
    }.flowOn(Dispatchers.IO)
}
```

This is the same as the **`fetchMovies()`** function, but this function uses Kotlin Flow and will return **`Flow<List<Movie>>`** to the function or class that will collect it. The Flow will emit the list of movies from **`movieService.getMovies`**, and it will flow on the **`Dispatchers.IO`** dispatcher.

3. Open the **`MovieViewModel`** class, and replace the initialization of the **`movies LiveData`** that gets the value from **`movieRepository`** with the following lines:

```
private val _movies =
  MutableStateFlow(emptyList<Movie>())
val movies: StateFlow<List<Movie>> = _movies
```

This will allow you to use the value of the **`_movies MutableStateFlow`** as the value of the **`movies StateFlow`**, which you will change later when you have fetched the list of movies from the Flow in **movieRepository**.

4. Do the same for the **error LiveData**, and replace its initialization with the value from **movieRepository** with the following lines:

```
private val _error = MutableStateFlow("")
val error: StateFlow<String> = _error
```

This will use the value of the **_error MutableStateFlow** for the **error StateFlow**. You will be able to change the value of this **StateFlow** later for handling the cases when the Flow encountered an exception.

5. Replace the **loading** and **_loading** variables with the following lines:

```
private val _loading = MutableStateFlow(true)
val loading: StateFlow<String> = _loading
```

This will use the value of the **_loading MutableStateFlow** for the **loading StateFlow**. You will update this later to indicate that the loading of movies is ongoing.

6. Remove the **fetchMovies()** function and its content. You will be replacing this in the next step.
7. Add a new **fetchMovies()** function that will collect the Flow from the **movieRepository.fetchMoviesFlow**, as shown in the following code block:

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

This will collect the list of movies from **`movieRepository.fetchMoviesFlow`** and set it to the **`_movies`** **`MutableStateFlow`** and the **`movies`** **`StateFlow`**. This list of movies will then be displayed in **`MainActivity`**.

8. Open the **`app/build.gradle`** file. Add the following lines in the dependencies:

```
implementation 'androidx.lifecycle:lifecycle-
runtime-ktx:2.4.1'
```

This will allow us to use **`lifecycleScope`** for collecting the flows in **`MainActivity`** later.

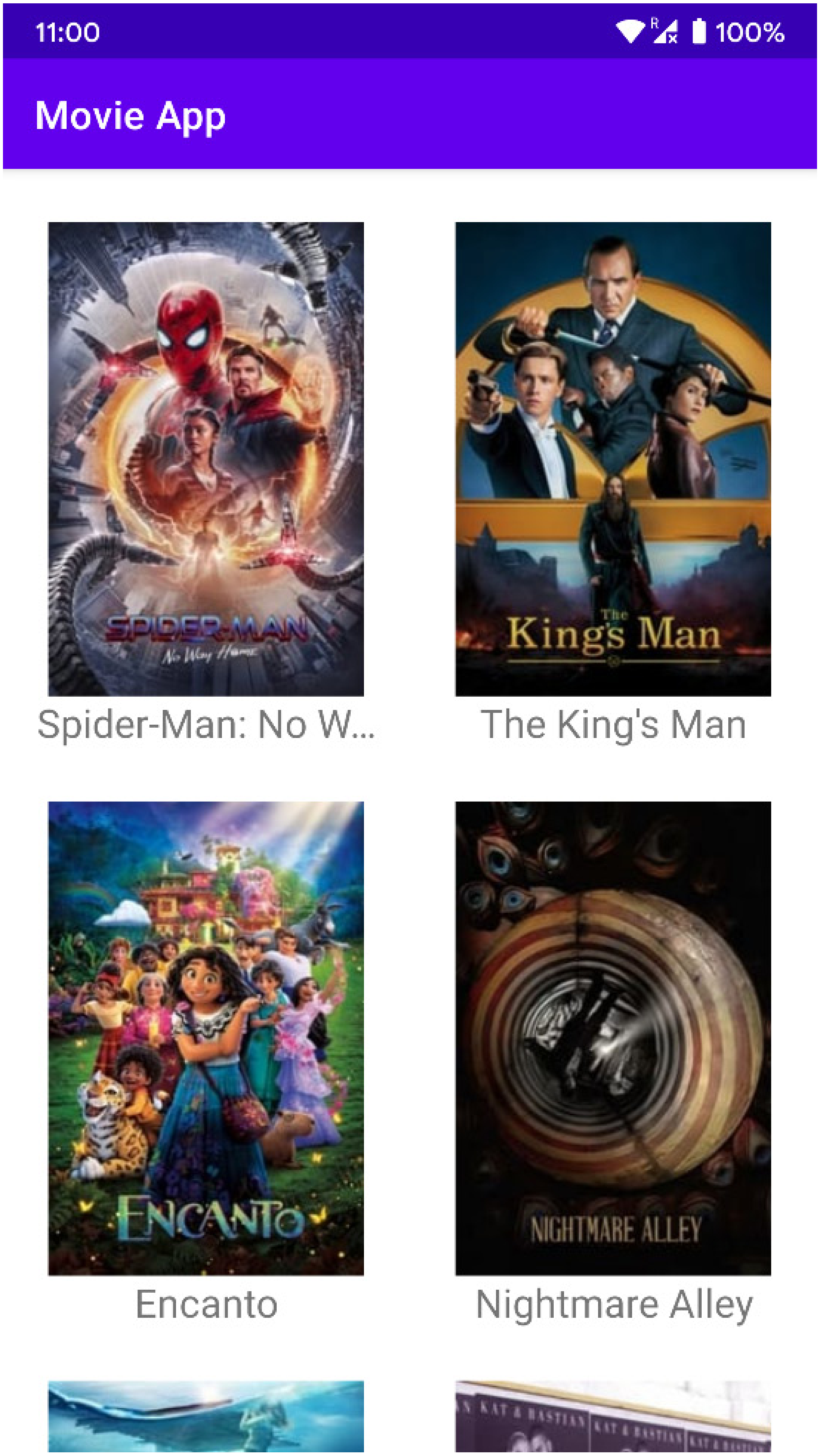
9. Open **`MainActivity`** and remove the lines of code that observe for the **`movies`**, **`error`**, and **`loading`** **`LiveData`**. Replace them with the following:

```
lifecycleScope.launch {
   repeatOnLifecycle(Lifecycle.State.STARTED) {
       launch {
           movieViewModel.movies.collect { movies -
>
               movieAdapter.addMovies(movies)
           }
       }
       launch {
           movieViewModel.error.collect { error ->
               if (error.isNotEmpty())
                 Snackbar.make(recyclerView, error,
                 Snackbar.LENGTH_LONG).show()
           }
       }
       launch {
           movieViewModel.loading.collect { loading
->
               progressBar.isVisible = loading
           }
       }
```

```
        }
    }
```

This will collect **`movies`** and add them to the list, collect the **`error`** and display a **`SnackBar`** message if **`error`** is not empty, and collect **`loading`** and update **`progressBar`** based on its value.

10. Run the application. The app should still display a list of movies (with a poster and a title), as shown in the following screenshot:


11:00
100%
Movie App
Spider-Man: No W...
The King's Man
Encanto
Nightmare Alley

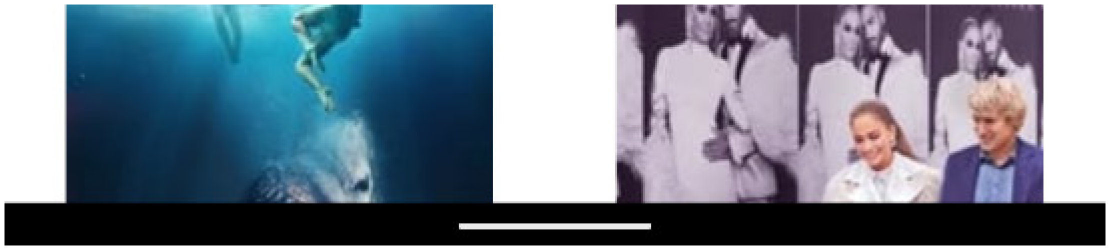


Figure 5.1 – The movie app with the list of movies

In this exercise, we have added Kotlin Flow in an Android app by creating a **`MovieRepository`** function that returns the list of movies as a Flow. This Flow was then collected by **`MovieViewModel`**.

Summary

This chapter focused on using Kotlin Flows for asynchronous programming in Android. Flows are built on top of Kotlin coroutines. A flow can emit multiple values sequentially, instead of just a single value.

We started with learning about how to use Kotlin Flows in your Android app. Jetpack libraries such as Room and some third-party libraries support Flow. To safely collect flows in the UI layer and prevent memory leaks and avoid wasting resources, you can use **`Lifecycle.repeatOnLifecycle`** and **`Flow.flowWithLifecycle`**.

We then moved on to creating Flows with Flow builders. The **`flowOf`** function creates a Flow that emits the value or **`vararg`** values you provided. You can convert collections and functional types to Flow with the **`asFlow()`** extension function. The **`flow`** builder function creates a new Flow from a suspending lambda block, inside which you can send values with **`emit()`**.

Then, we explored Flow operators and learned how you can use them with Kotlin Flows. With terminal operators, you can start the collection of the Flow. Intermediate operators allow you to transform a Flow into another Flow.

We then learned about buffering and combining Flows. With the **`buffer`**, **`conflate`**, and **`collectLatest`** operators, you can buffer Flows. You can combine Flows with the **`merge`**, **`zip`**, and **`combine`** Flow operators.

We then explored **`SharedFlow`** and **`StateFlow`**. These can be used in your Android projects. With **`SharedFlow`**, you can emit values to multiple listeners. **`StateFlow`** is **`SharedFlow`** that only emits the latest value to its listeners.

Finally, we worked on an exercise to add Kotlin Flows to an Android application. We used a Flow in **`MovieRepository`**, which was then collected in **`MovieViewModel`**.

In the next chapter, we will focus on how to handle Kotlin Flows cancelations and exceptions in your application.