

Chapter 11: Creating Infinite Lists with Jetpack Paging and Kotlin Flow

In the previous chapters, we built the great Restaurants App that displayed content from our own backend. However, the number of restaurants displayed in the Restaurants App was fixed, and the user was only able to browse through the few restaurants that we added to our Firebase database.

In this chapter, we will understand how pagination can help us display large datasets of items without putting pressure on our backend and without huge network bandwidth consumption. We will create the impression of an infinite list of items inside a new app that we will be working on called the Repositories App, and we will achieve that with the help of yet another Jetpack library called **Paging**.

In the first section, *Why do we need pagination?*, we will explore what data pagination is and how it can help us break large datasets into pages of data, thereby optimizing the communication between our app and the backend server. Up next, in the *Importing and exploring the Repositories App* section, we will explore a project in which we will integrate pagination: the Repositories App that displays information about GitHub repositories.

Then, in the *Using Kotlin Flow to handle streams of data* section, we will cover how paginated content can be expressed as a data stream and how Kotlin Flow is a great solution to handle such content. In the last section, *Exploring pagination with Jetpack Paging*, we will first explore the Jetpack Paging library as a solution to working with paginated content in our Android app, and then, with the help of this new library, we will integrate

paging in our Repositories App to create the illusion of an infinite list of repositories.

To summarize, in this chapter, we will be covering the following sections:

- Why do we need pagination?
- Importing and exploring the Repositories App
- Using Kotlin Flow to handle streams of data
- Exploring pagination with Jetpack Paging

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

Technical requirements

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

- The Arctic Fox 2020.3.1 version of Android Studio. You can also use a newer Android Studio version or even Canary builds but note that IDE interface and other generated code files might differ from the ones used throughout this book.
- Kotlin 1.6.10 or newer plugin installed in Android Studio
- The existing Repositories App from the GitHub repository of the book

The starting point for this chapter is represented by the Repositories App that you can find by navigating to the **Chapter_11** directory of the GitHub repository of the book, and then by importing the **repositories_app_starting_point_ch11** directory from within Android Studio. Don't worry, as we will do this together later in this chapter.

To access the solution code for this chapter, navigate to the **Chapter_11** directory and then import the **repositories_app_solution_ch11** directory from within Android Studio.

You can find the **Chapter_11** directory by following this link:

https://github.com/PacktPublishing/Kickstart-Modern-Android-Development-with-Jetpack-and-Kotlin/tree/main/Chapter_11

Why do we need pagination?

Let's say we have an Android application that allows you to explore GitHub repositories by displaying a list of projects. It does that by querying the GitHub **REpresentational State Transfer (REST) application programming interface (API)** with Retrofit and obtaining a fixed number of repositories inside the app. While the REST API serves the application with detailed information for each repository, the app only uses and displays the title and description of the repository.

NOTE

Don't confuse the Repository classes in our project architecture that abstract data logic with the GitHub repositories that are displayed in our Repositories App.

Now, let's imagine that this application retrieves and displays 20 repository elements. Because of this, the user will be able to scroll the content until the 20th element, and therefore will be able to visualize no more than 20 elements.

But what if we wanted to allow the user to explore more repositories inside our list? In the end, the purpose of the app is to browse a larger number of repositories and not just 20.

We could update the network call and request a larger list of elements from one single shot. In other words, we could refactor our app to obtain and display a list of 10,000 repositories on one occasion—that is, when the app is launched.

However, with such an approach, we can think of three main issues, as outlined here:

- **The user interface (UI) of the app could become unresponsive**—If our app tried to render all 10,000 elements, our UI would most likely freeze and become sluggish. However, this issue can be avoided by reusing or rendering only items that are visible on the screen. In fact, until now, we used the **LazyColumn** composable to render UI elements in a lazy manner (when needed), so we can conclude that this issue can be easily fixed.
- **The app would put a lot of pressure on the backend**—Imagine what would happen if every Android application client requested 10,000 database records from the backend server—these services would have to consume quite some resources to query and return so many elements.
- Such a **HyperText Transfer Protocol (HTTP)** request and response would cause a high network bandwidth consumption caused by the huge **JavaScript Object Notation (JSON)** payload that would have to be transferred. All 10,000 elements could contain a lot of fields and nested information—it's clear that having such a payload sent around between our apps and the server would be highly inefficient.

While we can easily address the first issue, we can conclude that the second and third issues are very concerning. Many real-world applications and systems face these problems, and in order to alleviate them, the concept of pagination was adopted for most client-server communication-based relationships where large datasets had to be displayed to the end user.

Pagination is a server-friendly communication approach that breaks a huge result into multiple smaller chunks. In other words, if your backend supports pagination, your application can request only a portion of data (often called a **page**) and receive a partial response, thereby allowing the transfer to be faster and more efficient on both sides.

When the application needs more results, it just requests another page, and another page, and so on. This approach is beneficial both for the app and backend service since only small portions of data are served and interpreted at a certain moment in time.

With pagination, if the user decides to visualize only a small portion of items and then switch to another app, your app would have requested only this small portion of data. Without pagination, in the same case, your backend would have served your app with the entire collection of items, while some of your users wouldn't have had a chance to see all of them. This would be a waste of resources from the perspective of your app, but especially from the perspective of your backend service. Also, only a small portion out of the huge payload sent over the internet was needed.

To implement such a pagination behavior on the UI, there are two well-known UI approaches for mobile apps, as follows:

- A fixed number of items are displayed on a screen that resembles a web page. On this page, there is a fixed amount of scrolling space because if the user wants to see new items, a button must be pressed to switch pages (often representing the number of a specific page), and then a new set of data is loaded and displayed, replacing the existing content.

From a mobile **user experience (UX)** perspective, this is a poor design choice because, as opposed to monitor screens used for web pages, scrolling over contents is more natural on smaller-sized devices such as phones.

- The list of items displayed grows as the user scrolls, thereby creating the impression that the list is infinite—such an approach is often referred to as infinite scrolling. While there is no such thing as an infinite list, this approach mimics one. It starts with a few requests for the initial page/s, and as the user scrolls to see more elements, it requests

more pages with more content on the fly. This approach relies heavily on scrolling and usually creates a better UX.

In this chapter, we will go for the second option—in other words, we will implement paging in an attempt to mimic the infinite list effect. Let's also try to visualize how the app could request more items as the user scrolls in the following simplified example, where **Page 1** contains only six elements:

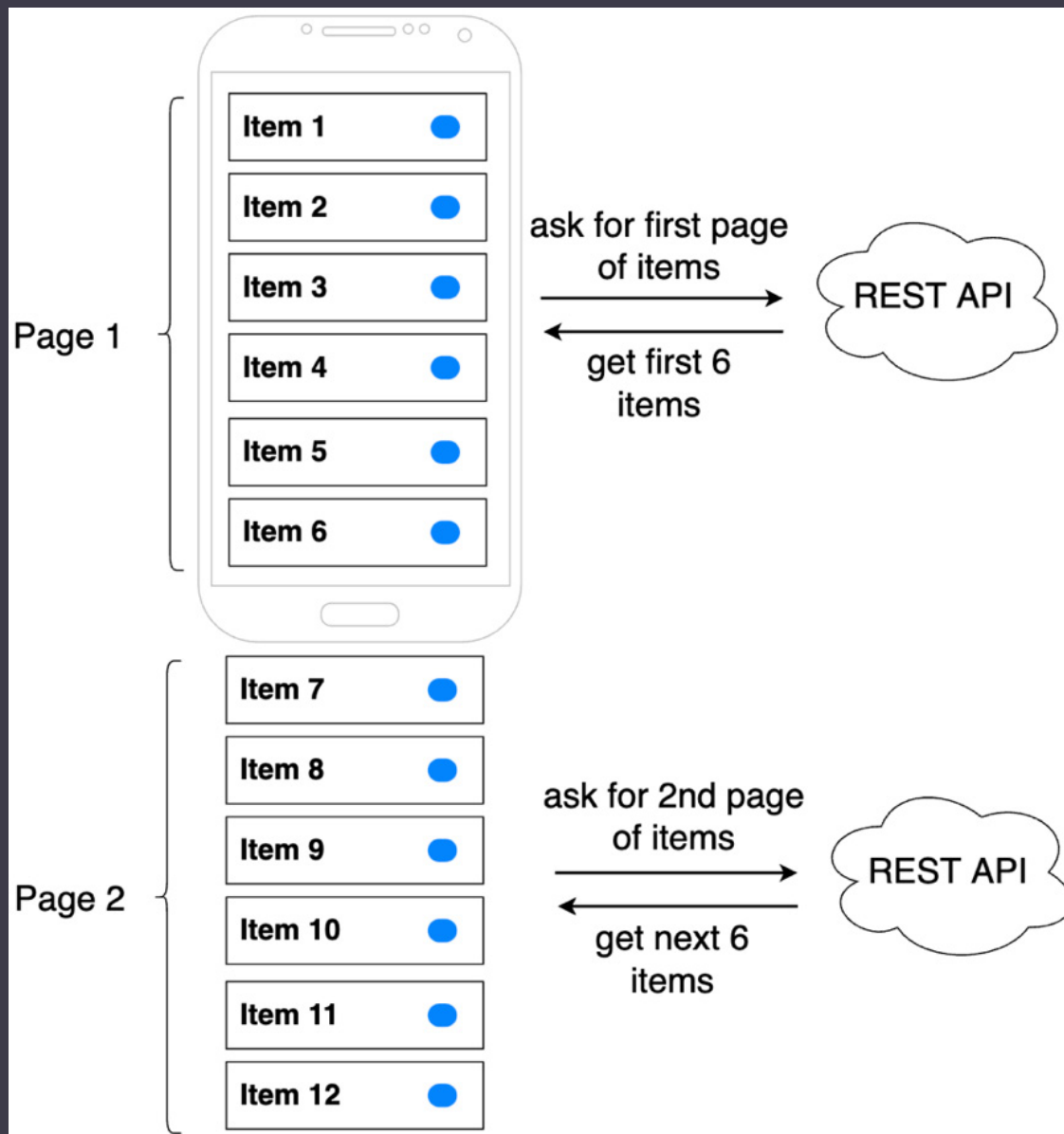


Figure 11.1 – Observing how infinite lists can be achieved with pagination

For the app to request the second page with items, the users must scroll further down, thereby informing the app about their intention of wanting

to see more elements.

When the app catches on to this intention (because the user reached the end of the list), it asks for the second page with items from the backend, making the list of repositories grow and allowing the user to browse through the new content. This process repeats on and on, as the user keeps on reaching the end of the list.

Before implementing this pagination approach, let's first get to know our starting point—the GitHub Repositories App!

Importing and exploring the Repositories App

The Repositories App project is a simple application that displays a list of repositories obtained from the GitHub Search API. This project is a simplified version of a Compose-based application that incorporates only a few concepts from the previous chapters as it tries to be a good candidate for implementing pagination with the Jetpack Paging library rather than being a fully-fledged sample app that applies all the concepts taught in the book.

Nevertheless, we will see how the Repositories App follows a **Model-View-ViewModel (MVVM)** presentation pattern, uses Retrofit to obtain data, a **ViewModel** class to hold state and present data, coroutines for the **asynchronous (async)** operation of obtaining data from the server, and Compose for the UI layer.

Let's start off by importing this project into Android Studio, as follows:

1. Navigate to the GitHub repository page of the book, located at <https://github.com/PacktPublishing/Kickstart-Modern-Android-Development-with-Jetpack-and-Kotlin>.

2. Download the repository files. You can do that by pressing the **Code** button and then by selecting **Download zip**.
3. Unzip the downloaded files and remember the location where you did this.
4. Open Android Studio, press on the **File** tab option, and then select **Open**.
5. Search for the directory where you unzipped the project files. Once you have found it, navigate to the **Chapter_11** directory, select the **repositories_app_starting_point_ch11** directory, and press **Open**.
6. Run the application on your test device.

You should notice that our Repositories App displays a list of repositories, and the index of each repository item from the list is displayed on the left side, as illustrated in the following screenshot:

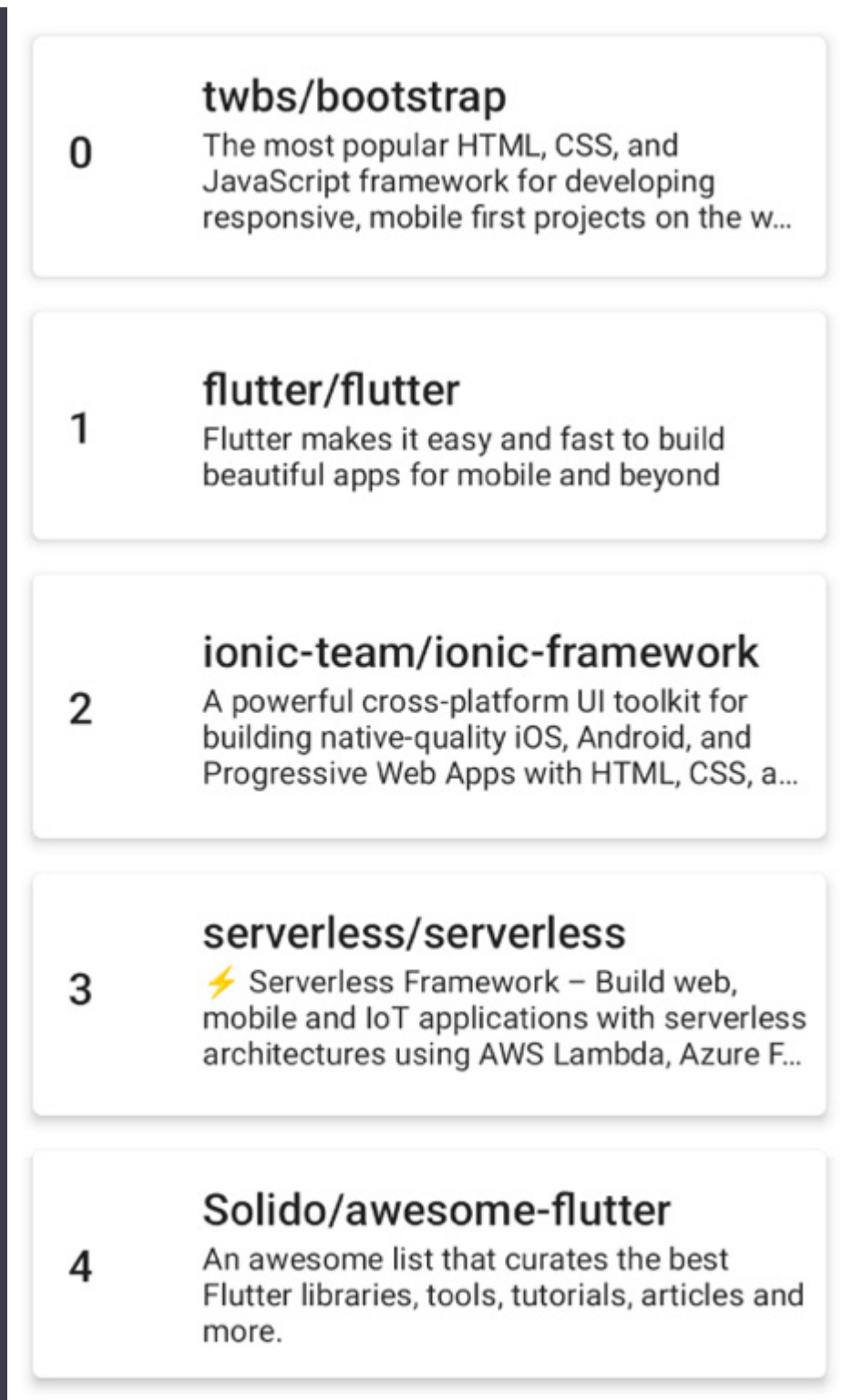


Figure 11.2 – Observing the Repositories App without pagination

If you scroll further down, you will notice that only 20 elements can be viewed. This means that our app doesn't support paging and the user can only browse through 20 repositories.

If we have a look inside the `RepositoriesApiService.kt` file, we will notice that our app instructs the REST API through the `@GET()` endpoint **Uniform Resource Locator (URL)** to obtain the first page of repositories while fetching only 20 items per page, as illustrated in the following code snippet:

```
interface RepositoriesApiService {  
    @GET("repositories?  
q=mobile&sort=stars&page=1&per_page=20")  
    suspend fun getRepositories():  
    RepositoriesResponse  
}
```

If you have a look at the parameters hardcoded within the request, you will notice that our app always requests the first page of repositories. Also, because it can specify the page number, this clearly means that the backend we're accessing supports pagination, but because we always ask for page **1**, our app doesn't take advantage of it.

More specifically, when the app performs this request, it will always retrieve 20 records from the backend from the page with index **1**. Later in this chapter, we will learn how to make multiple network calls requesting different page numbers, therefore adopting pagination.

NOTE

If you're looking to build an app that supports pagination, you must first make sure that your backend supports pagination, just as the GitHub Search API does. Remember that the whole purpose of pagination is to ease the workload of the backend API and to minimize the network bandwidth consumption associated with retrieving a huge JSON payload, so if your backend doesn't support pagination, you can't implement pagination in your app.

Let's have a brief look over the response we receive from the GitHub API by navigating to the `Repository.kt` file. Basically, we get a list of

Repository objects, and we parse the **id**, **name**, and **description** values of the repository, as illustrated in the following code snippet:

```
data class RepositoriesResponse(  
    @SerializedName("items") val repos:  
    List<Repository>  
)  
data class Repository(  
    @SerializedName("id")  
    val id: String,  
    @SerializedName("full_name")  
    val name: String,  
    @SerializedName("description")  
    val description: String)
```

As mentioned before, our app makes use of the GitHub Search API, and this can be better observed inside the **DependencyContainer.kt** class where the Retrofit **RepositoriesApiService** dependency is manually constructed, and the base URL of this API is passed. You can view the code for this process in the following snippet:

```
object DependencyContainer {  
    val repositoriesRetrofitClient:  
    RepositoriesApiService =  
        Retrofit.Builder()  
            .addConverterFactory(GsonConverterFactory  
            .create())  
            .baseUrl("https://api.github.com/search/")  
            .build().create(RepositoriesApiService::c  
lass.java)  
}
```

If you're looking to find out more about the API we're using in this chapter, head over to the official documentation of the GitHub Search API, at <https://docs.github.com/en/rest/search#search-repositories>.

Now, going back to our **Repositories App**, if we navigate to the **RepositoriesViewModel.kt** file, we will see that our **ViewModel** class uses the **RepositoriesApiService** dependency to obtain a list of repositories by launching a coroutine and setting the result to a Compose **State** object holding a list of **Repository** objects. The code is illustrated in the following snippet:

```
class RepositoriesViewModel(  
    private val restInterface: RepositoriesApiService  
        = DependencyContainer.repositoriesRetrofitClient  
) : ViewModel() {  
    val repositories =  
        mutableStateOf(emptyList<Repository>())  
    init {  
        viewModelScope.launch {  
            repositories.value =  
                restInterface.getRepositories().repos  
        }  
    }  
}
```

The approach of having a Jetpack **ViewModel** launch a coroutine to obtain data with the help of **Retrofit** is very similar to what we've done in the **Restaurants App**.

The UI level is also similar to the **Restaurants App**. If we navigate to the **MainActivity.kt** file, we can see that our **Activity** class creates a **ViewModel** instance, retrieves a Compose **State** object, obtains its value of type **List<Repository>**, and passes it to a composable function to consume it, as illustrated in the following code snippet:

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

```
RepositoriesAppTheme {  
    val viewModel: RepositoriesViewModel  
=  
    viewModel()  
    val repos =  
viewModel.repositories.value  
    RepositoriesScreen(repos)  
}  
}  
}
```

The composable function that consumes the list of **Repository** objects resides inside the **RepositoriesScreen.kt** file, as illustrated in the following code snippet:

```
@Composable  
fun RepositoriesScreen(repos: List<Repository>) {  
    LazyColumn(  
        contentPadding = PaddingValues(  
            vertical = 8.dp,  
            horizontal = 8.dp  
        ) {  
            itemsIndexed(repos) { index, repo ->  
                RepositoryItem(index, repo)  
            }  
        }  
    }  
}
```

Just as in the Restaurants App, our screen-level composable uses the **LazyColumn** composable to optimize the way the UI renders elements in the list.

LazyColumn usage is important for our use case of trying to implement pagination because we don't want our UI to render thousands of UI elements. Luckily, as we know already, **LazyColumn** has us covered because it only composes and lays out visible elements on the screen.

Now, you might have noticed that the **RepositoriesScreen** composable uses the **itemsIndexed()** **domain-specific language (DSL)** function instead of the **items()** function that we used in the Restaurants App. This is because, since our app will support pagination, we want to paint the index of the element displayed on the screen to better understand where we are at right now. To get the index of the composable item visible on the screen, the **itemsIndexed()** function provides us with this information out of the box.

Finally, let's have a brief look over the structure of the **RepositoryItem** composable that displays the contents of a **Repository** object, while also rendering the index of the repository, as follows:

```
@Composable
fun RepositoryItem(index: Int, item: Repository) {
    Card(
        elevation = 4.dp,
        modifier =
        Modifier.padding(8.dp).height(120.dp)
    ) {
        Row(
            verticalAlignment =
            Alignment.CenterVertically,
            modifier = Modifier.padding(8.dp)
        ) {
            Text(
                text = index.toString(),
                style = MaterialTheme.typography.h6,
                modifier = Modifier
                    .weight(0.2f)
                    .padding(8.dp))
            Column(modifier = Modifier.weight(0.8f))
        }
        Text(
            text = item.name,
```

```
                style =  
MaterialTheme.typography.h6)  
                Text(  
                    text = item.description,  
                    style =  
MaterialTheme.typography.body2,  
                    overflow = TextOverflow.Ellipsis,  
                    maxLines = 3)  
            }  
        }  
    }  
}
```

Now that we have briefly covered the current state of the Repositories App, we can conclude that it could really use pagination to show more repositories, especially when the GitHub Search API supports that. It's time to cover another important aspect that pagination forces us to be aware of, and that's the concept of streams of data.

Using Kotlin Flow to handle streams of data

If we want our app to support pagination in the form of an infinite list, it's clear that our existing approach of having a single one-shot request to the backend that results in one UI update will not suffice.

Let's first have a look in the following code snippet at how our `RepositoriesViewModel` class requests data:

```
class RepositoriesViewModel(  
    private val restInterface: RepositoriesApiService  
= [...] ) : ViewModel() {  
    val repositories =  
mutableStateOf(emptyList<Repository>())
```

```
init {  
    viewModelScope.launch {  
        repositories.value =  
            restInterface.getRepositories().repos  
    }  
}
```

When the **ViewModel** is initialized, it executes the **getRepositories()** suspending function inside a coroutine. The suspending function returns a list of **Repository** objects that is passed to the **repositories** variable. This means that our **ViewModel** performs a one-shot request for data in the form of a one-time call to the suspending function—no other request is done over time to get new repositories as the user scrolls through the list. That's why our app receives a single event with data (an initial list of objects) from the backend as a single result.

We can imagine that calling a similar **getRepositories()** suspending function with the one in our app would just as well return a one-time response as its return type would be **List<Repository>**, as illustrated in the following screenshot:

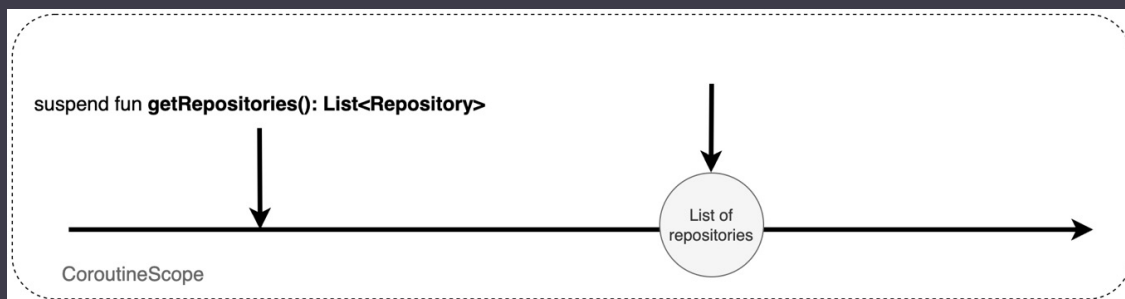


Figure 11.3 – Observing one-shot data result with suspending function

NOTE

*While our **ViewModel** contains a **repositories** variable whose type is **MutableState**, meaning that it can change its value over time, we aren't going to use **Compose State** objects to observe changes coming from the data layer as this would break the responsibilities of layers. Right now, in our code, we are calling a suspending function that returns only one result or*

*one set of data asynchronously. This result is passed to the **repositories** variable, so even though our UI state can change over time, it only receives one update.*

To support an infinite list, we must somehow design our app to receive multiple results over time, just as with a stream of data. In other words, our app must request new **Repository** objects as the user scrolls, thereby receiving multiple events with data, and not just one. With every new data event coming in, our app should get a new list of **Repository** objects that now contains the newly received repositories as well.

To make our **ViewModel** receive multiple events of data in the form of a stream of data, we can use Flow. Kotlin **Flow** is a data type built on top of coroutines that exposes a stream of multiple, asynchronously computed values.

As opposed to suspending functions, which emit a single result, Flow allows us to emit multiple values sequentially over time. However, just as a suspending function emits a result in an asynchronous manner that you can later obtain from within a coroutine, Flow also emits results asynchronously, so you must observe its results from within a launched coroutine.

You could use Flow to listen for events coming from various sources; for example, you could use Flow to get location updates every time the location of the user changes. Or, you could use Flow to get sequential updates from your Room database—instead of manually querying the database every time you insert or update items, you can tell Room to return a flow that will emit updates with its most up-to-date content whenever you perform insertions, updates, and so on.

Getting back to our example with repositories, let's imagine that our **getRepositories()** function is no longer a suspending function, but instead returns a flow whose contained data is of type **List<Repository>**, as illustrated in the following screenshot:

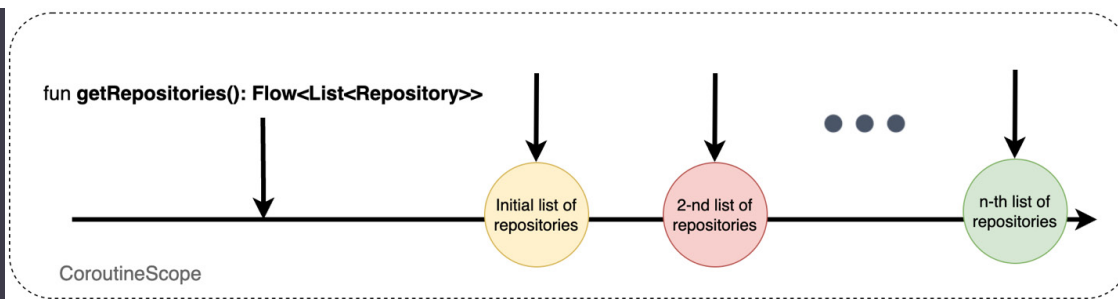


Figure 11.4 – Observing multiple results over time with Kotlin Flow

Just as the Compose **State** object holds data of a certain type (for example, **State<Int>** emits values of type **Int**), Flow also holds data of a certain type; in our previous example, that type was the data we're interested in emitting—that is, **List<Repository>**.

But how can we observe the emitted values of a flow?

Let's take the previous example where the **getRepositories()** method returned a **Flow<List<Repository>>** instance, and let's imagine that we're trying to observe its values in a UI component, as follows:

```
class SomeViewModel(...) : ViewModel() {
    init {
        viewModelScope.launch {
            getRepositories().collect { repos ->
                // Update UI
            }
        }
    }
    [...]
}
```

Since a flow emits values asynchronously, we obtained the **Flow<List<Repository>>** instance inside a launched coroutine and then called the **.collect()** method, which in turn provided us with a block of code where we can consume the **List<Repository>** values.

As opposed to obtaining such a list from a suspending function call, it's important to remember that the values emitted by the flow change (or, at

least, should change) over time. In other words, for every callback that provides us with a value stored in the `repos` variable, the content of its value of type `List<Repository>` could be different, allowing us to update the UI on every new emission.

In this section, we have explored what a flow is and how we can consume it. However, Kotlin Flow is a very complex subject; for example, we aren't going to cover the manner in which you can create a flow, or how you can modify the produced stream. If you're looking to find out more about Flow, check the official Android documentation at <https://developer.android.com/kotlin/flow>.

Let's now explore the last missing piece of the puzzle—the Paging library.

Exploring pagination with Jetpack Paging

To implement an infinite list of repositories in our Repositories App, we must find a way to request more repositories as the user scrolls through the existing list and reaches its bottom, thereby adding new elements on the fly. Instead of manually deciding when the user is approaching the bottom of the current list of repositories and then triggering a network request to get new items, we can use the Jetpack Paging library, which hides all this complexity from us.

Jetpack Paging is a library that helps us load and display pages of data from a large set of data, either through network requests or from our local data storage, thereby allowing us to save network bandwidth and optimize the usage of system resources.

In this chapter, for simplicity, we will use the Paging library to display an infinite list of repositories obtained from a network source (that is, the GitHub Search API), without involving the local cache.

NOTE

The Jetpack Paging library is now at its third implementation iteration, which is often referred to as Paging 3 or Paging v3. In this chapter, we will be using this latest version, so even though we will simply call it Jetpack Paging, we are in fact referring to Jetpack Paging 3.

The Jetpack Paging library abstracts most of the complexity associated with requesting the correct page at the correct time, depending on the scroll position of the user. Practically, it brings a lot of benefits to the table, such as the following:

- Avoidance of data request duplication—your app will request data only when needed; for example, when the user reaches the end of the list and more items must be rendered.
- Paged data is cached in memory out of the box. During the lifetime of the app process, once a page was loaded, your app will never request it again. If you cache the paginated data in a local database, then your application will not need to request a specific page for cases such as after an app restart.
- Paginated data is exposed as a data stream of the type that fits your need: Kotlin Flow, LiveData, or RxJava. As you might have guessed, we will use Flow.
- Out-of-the-box support for View System or Compose-based UI components that request data automatically when the user scrolls toward the end of the list. With such support, we don't have to know when to request new pages with data as the UI layer will trigger that for us out of the box.
- Retry and refresh capabilities triggered directly by the UI components.

Before moving to the actual integration of the Paging library, let's spend a bit of time looking over the main components part of the Paging API. To ensure paging in your application with the Jetpack Paging API, you must use the following:

- A **PagingSource** component—Defines the source of data for the paginated content. This object decides which page to request and loads it from your remote or local data source. If you're looking to have both a local and remote data source for your paginated content, you could use the built-in **RemoteMediator** API of the Paging library. Check out the *Further reading* section for more information on this.
- A **Pager** component—Based on the defined **PagingSource** component, you can construct a **Pager** object that will expose a stream of **PagingData** objects. You can configure the **Pager** object by passing a **PagingConfig** object to its constructor and specifying the page size of your data, for example.

The **PagingData** class is a wrapper over your paginated data containing a set of items part of the corresponding page. The **PagingData** object is responsible for triggering a query for a new page with items that is then forwarded to the **PagingSource** component.

- A dedicated UI component that supports pagination—To consume the stream of paginated content, your UI must make use of dedicated UI components that can handle paginated data. If your UI is based on the traditional View System, you could use the **PagingDataAdapter** component. Since our UI layer is based on Compose, **LazyColumn** has us covered as it knows how to consume paginated data (more on that in the next section).

To get a visual understanding of how all these components should fit, let's take the following example of a possible implementation of the Paging library inside our Repositories App:

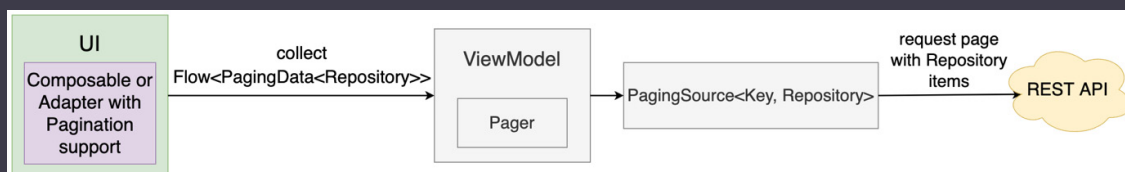


Figure 11.5 – Observing how Paging library APIs can be used in the Repositories App

At the UI level, our composable collects a flow that contains a stream of **PagingData<Repository>** objects. The **PagingData** object contains a list of **Repository** objects, and behind the scenes, it's responsible for forwarding requests for new pages to **PagingSource**, which in turn asks for new items from our REST API.

Inside **ViewModel**, we will have a **Pager** object that will use an instance of **PagingSource**. We will define a **PagingSource** object so that it knows which page to ask for and where to ask for it—that is, the GitHub Search API.

Now that we have covered the theoretical aspects of our pagination integration with Jetpack Paging, let's see which practical tasks we will be working on in this section. We will be doing the following:

- Implementing pagination with Jetpack Paging
- Implementing loading and error states plus retry functionality

Let's proceed with the first task: integrating pagination in our Repositories App.

Implementing pagination with Jetpack Paging

In this section, we will integrate paging in our Repositories App and create an infinite list of repositories with the help of Jetpack Paging. To achieve that, we will implement and add all the components described in the previous section.

Let's get cracking! Proceed as follows:

1. First, inside the app-level **build.gradle** file, in the **dependencies** block, add the Compose Gradle dependency for Jetpack Paging, as follows:

```
dependencies {  
    [...]  
    implementation "androidx.paging:
```

```
        paging-compose:1.0.0-alpha14"
    }
```

After updating the **build.gradle** file, make sure to sync your project with its Gradle files. You can do that by pressing on the **File** menu option and then by selecting **Sync Project with Gradle Files**.

2. Next up, let's refactor our Retrofit **RepositoriesApiService** interface by removing the hardcoded page index of **1** within the **@GET()** request annotation, and by adding a query **page** parameter of type **Int** representing the page index we're looking to acquire. The code is illustrated in the following snippet:

```
interface RepositoriesApiService {
    @GET("repositories?q=mobile&sort=stars&per_page=20")
    suspend fun getRepositories(@Query("page")
        page: Int):
        RepositoriesResponse
}
```

Before these changes, we were always obtaining the first page of repository results. Now, we have updated our network request to harvest the power of paginated REST APIs—that is, the capability to ask for a different page index based on the scrolling position of the user.

To achieve this, we used the Retrofit **@Query()** annotation, which basically will insert the value of the **page** parameter we have defined in the **getRepositories()** method into the **GET** request. As the GitHub Search API expects a **"page"** query key in the URL request, we have passed the **"page"** key to the **@Query()** annotation.

3. It's now time to build a **PagingSource** component that will request new pages through our **RepositoriesApiService** dependency and will keep track of which page to ask for, while also keeping an in-memory cache of the previously retrieved pages.

Inside the root package of the app, create a new class named **RepositoriesPagingSource** and paste the following code below it:

```
class RepositoriesPagingSource(  
    private val restInterface: RepositoriesApiService  
    = DependencyContainer.repositoriesRetrofitClient,  
    ) : PagingSource<Int, Repository>() {  
    override suspend fun load(params:  
LoadParams<Int>)  
        : LoadResult<Int, Repository> {  
    }  
    override fun getRefreshKey(  
        state: PagingState<Int, Repository>,  
    ): Int? {  
        return null  
    }  
}
```

Let's break down the code we have just added. This component is doing the following:

- It is in charge of requesting new pages, so it has a dependency on **RepositoriesApiService** as the **restInterface** constructor field.
- It is a **PagingSource** component, so it inherits from the **PagingSource** class while also defining the following:
 - A key as the type of the page index—in our case, the GitHub Search API requires an integer representing the index of the page, so we set the key as **Int**.
 - Type of the loaded data—in our case, **Repository** objects.
 - Implementing the following two mandatory functions:
 - The **load()** suspending function, which is called automatically by the Paging library and should fetch more items asynchronously. This method takes in a **LoadParams** object that keeps track of information such as what is the key (index) of the page that must be requested, or

the initial load size of items. Also, this method returns a **LoadResult** object indicating if a specific query result was successful or has failed.

- The **getRefreshKey()** function, which is called to obtain and return the most recent page key in case of a refresh event so that the user is returned to the latest known page (and not the first one). A refresh event can come from a variety of sources, such as a manual UI refresh triggered by the user, a database cache invalidation event, system events, and so on.

For simplicity, and also because we will not implement refresh capabilities, we will skip implementing the **getRefreshKey()** method, so we just returned **null** inside the body of this method. However, if you're looking to also support such behavior, check out the *Further reading* section where additional resources are listed to help you provide an implementation for this method.

4. Now that we have covered the purpose of the two mandatory methods, let's implement the one we're really interested in—the **load()** function.

This method should return a **LoadResult** object, so first, add a **try-catch** block, and inside the **catch** block, return an **Error()** instance of **LoadResult** by passing the **Exception** object that was caught, as illustrated in the following code snippet:

```
class RepositoriesPagingSource(...) : [...] {  
    override suspend fun load(params:  
LoadParams<Int>)  
        : LoadResult<Int, Repository> {  
        try {  
        } catch (e: Exception) {  
            return LoadResult.Error(e)  
        }  
    }  
    override fun getRefreshKey(...): Int? { ... }
```

```
}
```

With this approach, if the request for a new page fails, we let the Paging library know that an error event occurred by returning the `LoadResult.Error` object.

5. Next up, inside the `try` block, we must first obtain and store the next page we're interested in. Store the index of the next page inside the `nextPage` variable, as follows:

```
class RepositoriesPagingSource(...) : [...] {  
    override suspend fun load(params:  
LoadParams<Int>)  
        : LoadResult<Int, Repository> {  
        try {  
            val nextPage = params.key ?: 1  
        } catch (e: Exception) {  
            return LoadResult.Error(e)  
        }  
    }  
    override fun getRefreshKey(...): Int? { ... }  
}
```

We obtained the index for the next page by tapping into the `params` parameter and getting its `key` field—this field will always give us the index of the next page that must be loaded. If this is the first time a page is requested, the `key` field will be `null`, so we default to the value of `1` in that case.

6. Since we now know the index of the next page of repositories that we need, let's query our REST API for that specific page by calling the `getRepositories()` method of `restInterface` and by passing in the newly defined `nextPage` parameter, as follows:

```
class RepositoriesPagingSource(...) : [...] {  
    override suspend fun load(params:  
LoadParams<Int>)  
        : LoadResult<Int, Repository> {
```

```

        try {
            val nextPage = params.key ?: 1
            val repos = restInterface
                .getRepositories(nextPage).repos
        } catch (e: Exception) {
            return LoadResult.Error(e)
        }
    }
    override fun getRefreshKey(...): Int? { ... }
}

```

In this step, we also store a list of **Repository** objects from within the response inside the **reposResponse** variable.

7. Next up, we must return a **LoadResult** object, as the request to our REST API is successful at this point. Let's instantiate and return a **LoadResult.Page** object, as follows:

```

class RepositoriesPagingSource(...) : [...] {
    override suspend fun load(params:
LoadParams<Int>)
        : [...] {
        try {
            val nextPage = params.key ?: 1
            val repos = restInterface
                .getRepositories(nextPage).repos
            return LoadResult.Page(
                data = repos,
                prevKey = if (nextPage == 1) null
                    else nextPage - 1,
                nextKey = nextPage + 1)
        } catch (e: Exception) {
            return LoadResult.Error(e)
        }
    }
    override fun getRefreshKey(...): Int? { ... }
}

```

```
}
```

We had to pass the following to the `LoadResult.Page()` constructor:

- A list of **Repository** objects from the newly requested page to the **data** parameter.
- The previous key of the newly requested page to the **prevKey** parameter. This key is important if, for some reason, the previous pages are invalidated and must be reloaded when the user starts scrolling up. Most of the time, we would deduct **1** from the **nextPage** value, yet we also made sure that if we had just requested the first page (the value of **nextPage** would be **1**), we would pass **null** to the **prevKey** parameter.
- The next key after **nextPage** to the **nextKey** parameter. This is a simple one as we have just added **1** to the value of **nextPage**.

Now that we finished the **PagingSource** implementation, it's time to build the **Pager** component and get a stream of paginated data.

8. Inside **RepositoriesViewModel**, replace the **RepositoriesApiService** dependency with the newly created **RepositoriesPagingSource** class, as follows:

```
class RepositoriesViewModel(  
    private val reposPagingSource:  
        RepositoriesPagingSource =  
        RepositoriesPagingSource()  
): ViewModel() {  
}
```

At the same time, we make sure to remove any existing implementation inside the **RepositoriesViewModel**, leaving it blank for the upcoming step.

9. Still inside the **RepositoriesViewModel**, define a **repositories** variable that will hold our flow of paginated data, like this:

```
import kotlinx.coroutines.flow.Flow  
class RepositoriesViewModel(  
    private val reposPagingSource:  
        RepositoriesPagingSource =  
        RepositoriesPagingSource()  
): ViewModel() {  
    val repositories: Flow<List<Repository>> =  
        reposPagingSource.flow()  
}
```

```

        private val reposPagingSource:
        RepositoriesPagingSource =
        RepositoriesPagingSource()
    ) : ViewModel() {
        val repositories: Flow<PagingData<Repository>>
    }

```

The paginated content with **Repository** items is held within a **PagingData** container, making our stream of data to be of type **Flow<PagingData<Repository>>**.

Now, we must instantiate our **repositories** variable. However, creating a flow is not trivial, especially when the data (the list of repositories) must grow as the user scrolls. The Paging library has us covered, as it will hide this complexity from us and will provide us with a flow that emits data as we would expect it to: when the user scrolls to the end of the list, new requests are made to the backend, and new **Repository** objects are appended to the list.

10. As the first step to obtaining our flow of paginated data, we must create an instance of the **Pager** class based on the previously created **PagingSource** object, like so:

```

class RepositoriesViewModel(
    private val reposPagingSource:
    RepositoriesPagingSource =
    RepositoriesPagingSource()
) : ViewModel() {
    val repositories: Flow<PagingData<Repository>>
    =
        Pager(
            config = PagingConfig(pageSize = 20),
            pagingSourceFactory = {
                reposPagingSource
            })
}

```

To create an instance of a **Pager**, we called the **Pager()** constructor and passed the following:

- A **PagingConfig** object with a **pageSize** value of **20** (to match this value with the number of repositories we're requesting from the backend) to the **config** parameter.
- The **reposPagingSource** instance of type **RepositoriesPagingSource** to the **pagingSourceFactory** parameter. By doing so, the Paging library will know which **PagingSource** object to query for new pages.

11. Finally, to obtain a flow with data from the newly created **Pager** instance, we must simply access the **flow** field exposed by the resulted **Pager** instance, as follows:

```
class RepositoriesViewModel(...) : ViewModel() {  
    val repositories: Flow<PagingData<Repository>>  
    =  
        Pager(  
            config = PagingConfig(pageSize = 20),  
            pagingSourceFactory = {  
                reposPagingSource  
            }).flow.cachedIn(viewModelScope)  
}
```

On the resulting flow, we also called the **cachedIn()** extension function that makes sure that the stream of data is kept alive as long as the passed **CoroutineScope** object is alive and then returns back the same flow it's called upon. Since we wanted the paginated content to be cached as long as the **ViewModel** is kept in memory, we passed the **viewModelScope** scope to this extension function. This makes sure that the flow is also preserved upon events where the **ViewModel** survives—for example, configuration change.

12. Now, we must obtain the flow in our Compose-based UI, so inside the **RepositoriesAppTheme()** composable call from within **MainActivity**, re-

place the **repos** variable with the **reposFlow** variable that holds a reference to the **repositories** flow variable of the **ViewModel**, as follows:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            RepositoriesAppTheme {
                val viewModel:
RepositoriesViewModel =
                    viewModel()
                val reposFlow =
viewModel.repositories
                RepositoriesScreen()
            }
        }
    }
}
```

13. Next up, we must use a special collection function (similar to the **collect()** function used in the previous section) that can consume and remember the paginated data from within **reposFlow** in the context of Compose.

Declare a new variable called **lazyRepoItems** and instantiate it with the result returned from the **collectAsLazyPagingItems()** call on **reposFlow**, as follows:

```
class MainActivity : ComponentActivity() {
    override fun onCreate(savedInstanceState:
Bundle?) {
        super.onCreate(savedInstanceState)
        setContent {
            RepositoriesAppTheme {
                val viewModel: [...] = viewModel()
                lazyRepoItems = reposFlow.collectAsLazyPagingItems()
                RepositoriesScreen()
            }
        }
    }
}
```

```

        val reposFlow =
viewModel.repositories
        val lazyRepoItems
            : LazyPagingItems<Repository>
=
        reposFlow.collectAsLazyPagingItem
s()
        RepositoriesScreen(lazyRepoItems)
    }
}
}
}

```

The `collectAsLazyPagingItems()` function returned a `LazyPagingItems` object filled with `Repository` objects. The `LazyPagingItems` object is responsible for accessing `Repository` objects from the flow so that they can be consumed by our `LazyColumn` component later on—that's why, in the end, we passed `lazyRepoItems` to the `RepositoriesScreen()` composable.

14. Moving to the last piece of the puzzle, the `RepositoriesScreen()` composable, make sure that it accepts the `LazyPagingItems` object returned by our flow by adding the `repos` parameter, as follows:

```

@Composable
fun RepositoriesScreen(
    repos: LazyPagingItems<Repository>
) {
    LazyColumn (...) {
    }
}

```

Also, while you're at this step, remove all the code inside the DSL `content` block exposed by `LazyColumn` as we will re-add it in a different structure in the next step.

15. Finally, still inside `RepositoriesScreen()`, pass the `repos` input parameter to another `itemsIndexed()` DSL function that accepts

LazyPagingItems, as follows:

```
@Composable
fun RepositoriesScreen(
    repos: LazyPagingItems<Repository>
) {
    LazyColumn(...) {
        itemsIndexed(repos) { index, repo ->
            if (repo != null) {
                RepositoryItem(index, repo)
            }
        }
    }
}
```

The **LazyColumn** API knows how to consume paginated data and how to report back to our instances of **Pager** and **PagingSource** when a new page should be loaded, and that's why we made use of an overloaded variant of the **itemsIndexed()** DSL function that accepts **LazyPagingItems** as content.

Also, because the returned **repo** value can be **null**, we added a null check before passing it to our **RepositoryItem()** composable.

16. Finally, build and run the application. Try to scroll to the bottom of the repositories list. This should trigger a request to get new items, and therefore you should be able to scroll and browse through an *endless* list of repositories.

NOTE

If you make too many requests to the GitHub Search API, you might be temporarily limited, and the application will stop loading new items and throw an error. To make our application express such an event, we will learn how to display error states, up next.

Next up, let's improve the UI and UX of our application by adding loading and error states in the context of an infinite list.

Implementing loading and error states plus retry functionality

While our application now features an infinite list that the user can scroll through, it doesn't express any sort of loading or error state. The good news is that the Paging library tells us exactly when loading states or error states must be shown.

However, before jumping into the actual implementation, we should first cover the possible loading states and error states that emerge from interacting with an app that features pagination. Luckily, all these cases are already covered by the Paging API.

While the **LazyPagingItems** API provides us with several **LoadState** objects, the most common ones—and the ones we will need in this section—are the **refresh** and **append** types, as explained in more detail here:

- The **LoadState.refresh** instance of **LoadState** represents initial states that occur after the first request of paginated items or after a refresh event. The two values that we're interested in for this object are these:
 - **LoadState.Loading** — This state means that the app is expressing the initial loading status. When this status arrives for the first time after an app launch, no content would be painted on the screen at that point.
 - **LoadState.Error** — This state means that the app is expressing the initial error status. Just as with the previous state, if this status arrives for the first time after an app launch, no content is present.
- The **LoadState.append** instance of **LoadState** represents states that occur at the end of a subsequent request of paginated items. The two values we're interested in for this object are similar to type **refresh** but have different significance, as outlined here:
 - **LoadState.Loading** — This state means that the app is in a loading status at the end of a subsequent request for a page with repositories; in other words, the app has requested another page with

repositories and it's waiting for the results to arrive. At this point, there should be content rendered from the previous pages.

- **LoadState.Error** — This state means that the app reached an error status after a subsequent request for a page with repositories. In other words, the app has requested another page with repositories but the request has failed. Just as with the previous state, there should be content rendered from the previous pages.

Let's listen for these states in our app and start with type **LoadState.refresh**, as follows:

1. Inside the **RepositoriesScreen()** composable, below the **itemsIndexed()** call, store the **refresh** load state instance inside the **refreshLoadstate** variable, as follows:

```
@Composable
fun RepositoriesScreen(
    repos: LazyPagingItems<Repository>
) {
    LazyColumn(...) {
        itemsIndexed(repos) { index, repo ->
            if (repo != null) {
                RepositoryItem(index, repo)
            }
        }
        val refreshLoadState =
        repos.loadState.refresh
    }
}
```

Every time this refreshes, **LoadState** will change; the values within **refreshLoadState** will be the latest ones and will correspond to the page where they occurred.

2. Next up, create a **when** expression and verify whether **refreshLoadState** is of type **LoadState.Loading**, and if it is, inside a new **item()** call, pass a

LoadingItem() composable that we will define in a bit. The code is illustrated in the following snippet:

```
@Composable
fun RepositoriesScreen(
    repos: LazyPagingItems<Repository>
) {
    LazyColumn(...) {
        itemsIndexed(repos) { index, repo ->
            if (repo != null) {
                RepositoryItem(index, repo)
            }
        }
        val refreshLoadState =
            repos.loadState.refresh
        when {
            refreshLoadState is LoadState.Loading -
            > {
                item {
                    LoadingItem(
                        Modifier.fillParentMaxSize(
                    ))
                }
            }
        }
    }
}
```

Since we are adding another **item()** call below the **itemsIndexed()** DSL call, we are actually adding another composable below the list of composables from the **itemsIndexed()** call. However, since **refreshLoadState** can be of type **LoadState.Loading** on the first request for a page of items, this means that the screen is empty at this time, so we also passed a **fillParentMaxSize** modifier to the **LoadingItem()** composable, thus making sure that this composable will take up the entire size of the screen.

3. Next up, at the bottom of the `RepositoriesScreen.kt` file, let's quickly define a `LoadingItem()` function that will feature a `CircularProgressIndicator()` composable, as follows:

```
@Composable
fun LoadingItem(
    modifier: Modifier = Modifier
) {
    Column(
        modifier = modifier.padding(24.dp),
        verticalArrangement = Arrangement.Center,
        horizontalAlignment =
            Alignment.CenterHorizontally
    ) { CircularProgressIndicator() }
}
```

4. Now, run the app, and notice how the progress indicator animation is running before the first page of repositories is loaded and how it is occupying the entire screen, as illustrated in the following screenshot:

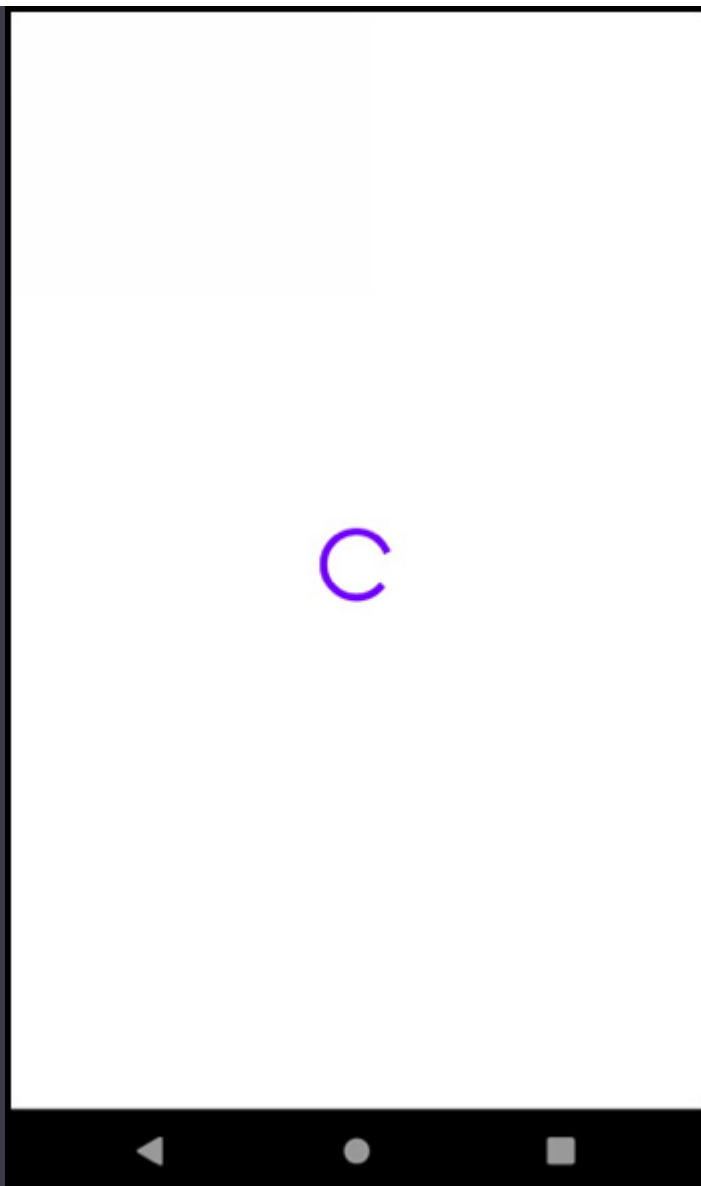


Figure 11.6 – Adding a loading animation for the first request of paginated content

5. Now, let's cover the case where `refreshLoadState` is of type `Loadstate.Error`. Back inside the `LazyColumn` component of the `RepositoriesScreen()` composable, below the first `when` branch, add another check for the state to be `LoadState.Loading`—and if that's the case, add an `ErrorItem()` composable that we will define in a bit. The code that you must add is illustrated in the following snippet:

```
@Composable
fun RepositoriesScreen(
    repos: LazyPagingItems<Repository>
) {
```

```

        LazyColumn(...) {
            itemsIndexed(repos) { index, repo -> [...] }
            val refreshLoadState =
                repos.loadState.refresh
            when {
                refreshLoadState is LoadState.Loading -
            > {
                item { LoadingItem(...) }
            }
                refreshLoadState is LoadState.Error ->
            {
                val error = refreshLoadState.error
                item {
                    ErrorItem(
                        message =
                            error.localizedMessage
                                ?: "",
                        modifier =
                            Modifier.fillParentMaxSize()
                    )
                }
            }
        }
    }
}

```

The **ErrorItem()** composable requires an error message to display, so we stored the **Throwable** object from **LoadState** in the **error** variable and passed its **localizedMessage** value to the **message** parameter of the composable.


Similar to the **LoadState.Loading** case from before, we are adding another **item()** call below the **itemsIndexed()** DSL call, so we are actually adding another composable below the list of composables from the **itemsIn-**

`dexed()` call. Also, since `refreshLoadState` can be of type `LoadState.Error` on the request for the first page of items, this means that the screen is empty at this time, so we also passed a `fillParentMaxSize` modifier to the `ErrorItem()` composable, thus making sure that this composable is taking up the entire size of the screen.

6. Next up, at the bottom of the `RepositoriesScreen.kt` file, let's quickly define an `ErrorItem()` function that will feature a `Text()` composable displaying a red error message, as follows:

```
@Composable
fun ErrorItem(
    message: String,
    modifier: Modifier = Modifier) {
    Row(
        modifier = modifier.padding(16.dp),
        horizontalArrangement =
            Arrangement.SpaceBetween,
        verticalAlignment =
            Alignment.CenterVertically
    ) {
        Text(
            text = message,
            maxLines = 2,
            modifier = Modifier.weight(1f),
            style = MaterialTheme.typography.h6,
            color = Color.Red)
    }
}
```

7. To mimic an error state, run the app on your emulator or physical device without an internet connection, and you should see a similar error occupying the entire screen, as illustrated in the following screenshot:



failed to connect to /10.0.2.2 (port
8000) from /169.254.61.43 (port
8000)

Figure 11.7 – Adding an error message for the first request of paginated content

Note that the error message could be different depending on the circumstances of the error scenario that you have created.

Before moving on to the append type of **LoadState**, let's briefly cover the retry functionality that is provided out of the box by the Paging library. In other words, we want to give the user the option to retry obtaining the data in case something went wrong, such as with our forced-error case of disconnecting the test device from the internet.

Let's do that next.

8. Refactor the **ErrorItem()** composable to accept an **onClick()** function parameter that will be triggered by the **onClick** event caused by the press of a new retry **Button()** composable, as follows:

```
@Composable  
fun ErrorItem(  
    onClick: () -> Unit = {}  
) {  
    // ...  
}
```

```

        message: String,
        modifier: Modifier = Modifier,
        onClick: () -> Unit) {
    Row(...) {
        Text(...)
        Button(
            onClick = onClick,
            modifier = Modifier.padding(8.dp)
        ) { Text(text = "Try again") }
    }
}

```

Also, inside the `Row()` composable that was displaying the error message, we have now added a `Button()` composable that when pressed, forwards the event to its caller.

9. Then, back inside the `LazyColumn` component of `RepositoriesScreen()`, find the case where `LoadState` is of type `LoadState.Error` and implement the `onClick` parameter of the `ErrorItem()` composable that will now trigger a reload. The code is illustrated in the following snippet:

```

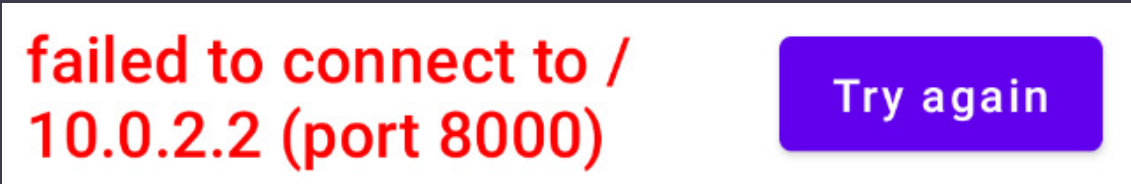
@Composable
fun RepositoriesScreen(
    repos: LazyPagingItems<Repository>
) {
    LazyColumn(...) {
        itemsIndexed(repos) { index, repo -> [...] }
        val refreshLoadState =
            repos.loadState.refresh
            when {
                refreshLoadState is LoadState.Loading -
            > {
                ...
            }
                refreshLoadState is LoadState.Error ->
            {

```

```
        val error = refreshLoadState.error
        item {
            ErrorItem(
                message =
                    error.localizedMessage
                        ?: "",
                modifier =
                    Modifier.fillParentMaxSize(),
                onClick = { repos.retry()
            })
        }
    }
}
```

To trigger a reload, we called the `retry()` function provided by our **LazyPagingItems** instance. Behind the scenes, when the `retry()` function is called, the Paging library notifies **PagingSource** to request the problematic page again—in this case, for us, the first page with repositories.

10. Run the app on your emulator or physical device without an internet connection. You should now see the error state occupying the entire screen containing the error message, but also a retry button. The following screenshot provides a depiction of this:



failed to connect to /
10.0.2.2 (port 8000)

Try again

Figure 11.8 – Adding error message and retry button for the first request of paginated content

Don't press the retry button just yet.

11. Reconnect your device to the internet and then press the retry button.
As an effect of this action, the content should now load successfully.

Now that we have covered the possible **LoadState** values for the **refresh** state, it's time to also cover the values for the **append** state. As we previously stated, type **LoadState.append** represents states that occur at the end of a subsequent request of paginated items.

The possible states we're interested in for this scenario are the **LoadState.Loading** state—meaning the user has scrolled toward the end of the list and the app is waiting for another page with repositories—and the **LoadState.Error** state—meaning that the user has scrolled toward the end of the list but the request to get a new page with repositories has failed.

12. Inside the block of code exposed by the **itemsIndexed()** call from within the **RepositoriesScreen()** composable, just as we did with the **refresh** state, store the **append** state inside a new **appendLoadState** variable, and then add two corresponding branches inside the **when** expression treating the **LoadState.Loading** and the **LoadState.Error** cases. The code is illustrated in the following snippet:

```
@Composable
fun RepositoriesScreen(
    repos: LazyPagingItems<Repository>
) {
    LazyColumn(...) {
        itemsIndexed(repos) { [...], }
        val refreshLoadState =
            repos.loadState.refresh
        val appendLoadState =
            repos.loadState.append
        when {
            refreshLoadState is LoadState.Loading -
            > {
                item {
                    LoadingItem(...)
                }
            }
        }
    }
}
```

```
        }
    }
    refreshLoadState is LoadState.Error ->
{
    val error = refreshLoadState.error
    item {
        ErrorItem(
            message =
error.localizedMessage
            ?: "",
            modifier = ...,
            onClick = { repos.retry()
        })
    }
}
appendLoadState is LoadState.Loading ->
{
    item {
        LoadingItem(
            Modifier.fillMaxWidth())
    }
}
appendLoadState is LoadState.Error -> {
    val error = appendLoadState.error
    item {
        ErrorItem(
            message =
error.localizedMessage
            ?: "",
            onClick = { repos.retry()
        })
    }
}
}
```

```
}
```

The way we treated the possible values of **appendLoadState** is very similar to how we treated the possible values of **refreshLoadState**. However, the notable difference is that **appendLoadState** state values occur when the app has already loaded some pages and the user has scrolled toward the end of our list, meaning that our app is either waiting for a new page with repositories or failed to load it.

That's why, in the **LoadState.Loading** case, we have passed the **Modifier.fillMaxWidth()** modifier to the **LoadingItem()** composable, therefore making sure that the loading indicator item appears at the bottom of the list as a list element. In other words, the loading element will take only the available width and it will not cover the entire screen like we did when **refreshLoadState** was of type **LoadState.Loading**.

Similarly, for the **LoadState.Error** case, we passed the **Modifier.fillMaxWidth()** modifier to the **ErrorItem()** composable, therefore making sure that the error element appears as a list element and doesn't cover the entire screen like we did when **refreshLoadState** was of type **LoadState.Error**.

Let's see these two cases in practice, and let's start with the case when our **appendLoadState** instance has a value of **LoadState.Loading**.

13. First, run the app while your test device is connected to the internet. If you scroll down to the bottom of the list with repositories, you should see the loading indicator animation displayed until a new page with repositories is loaded, as illustrated in the following screenshot:



Figure 11.9 – Adding a loading animation for a subsequent request of paginated content

Unlike the loading indicator that is shown initially, this indicator appears as an item within the list, thereby indicating that the app is waiting for a new page with repositories.

NOTE

*If your network speed is very fast, you might miss the loading spinner as you are scrolling through new pages. To simulate a slower connection, you can change the network speed of your Android emulator by going into **AVD Manager**, pressing the **Edit** button of your emulator, and then selecting **Show Advanced Settings**. Inside this menu, you can slow down the internet speed of your emulator so that you can see the loading spinner.*

Now, let's test the case when our **appendLoadState** instance is of type **LoadState.Error**.

14. First, run the app while your test device is connected to the internet.
15. Then, disconnect your test device from the internet and scroll down to the bottom of the list with repositories. Initially, you might see the loading indicator, yet after a short period of time, you should see the error element appearing at the bottom of the list, as illustrated in the following screenshot:

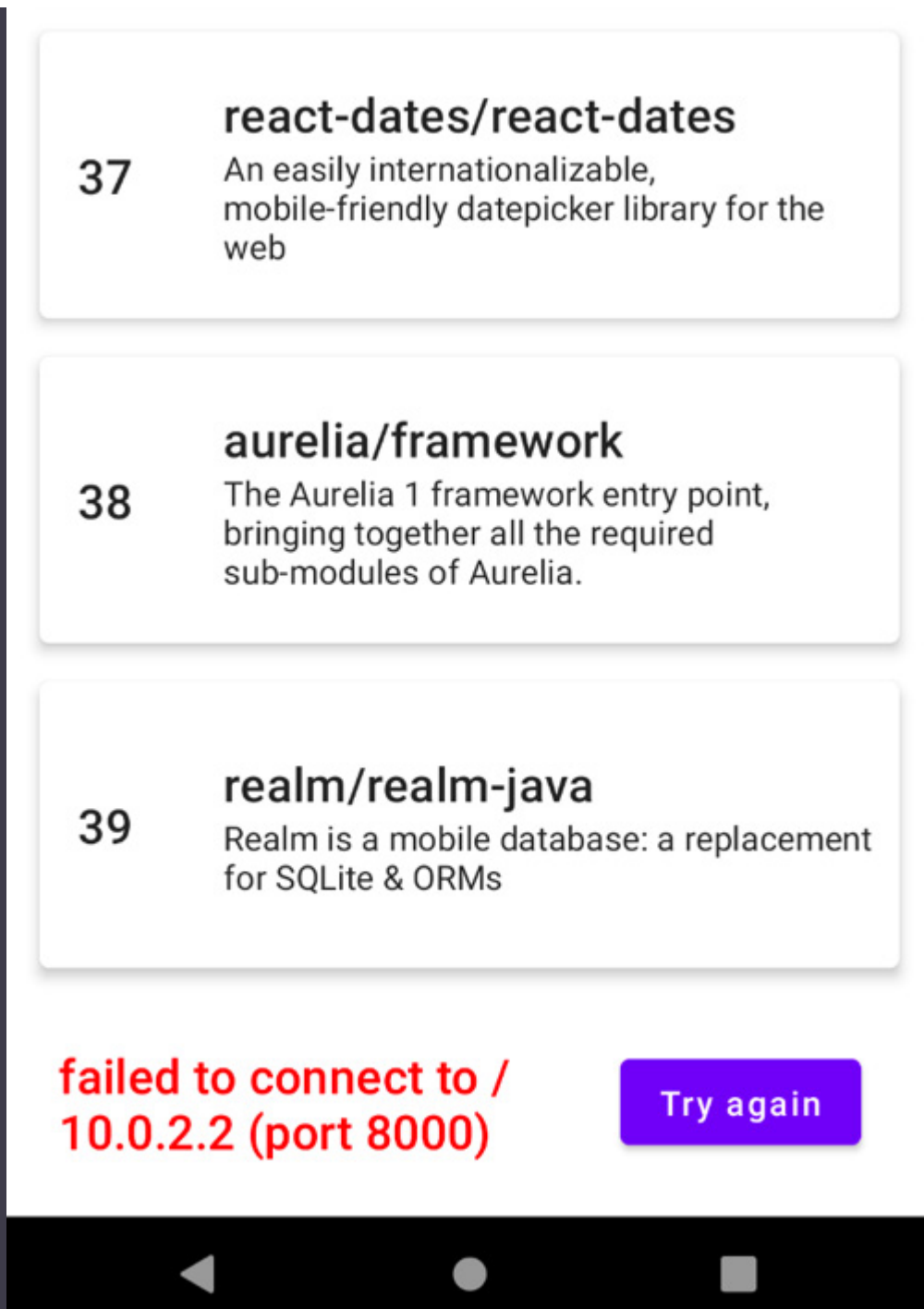


Figure 11.10 – Adding error element for a subsequent request of paginated content

Unlike the error message that is shown initially, this error element appears as an item in the list, thereby indicating that the app has failed to obtain the next page with repositories.

16. Optionally, you can reconnect your device to the internet and press the retry button—the new page with repositories should now load, so

you can continue browsing and scrolling for more items.

Summary

In this chapter, we first understood what pagination is and how pagination can be used to expose large datasets of items to users in a more efficient manner.

Then, we got to meet the Repositories App, a simple Android project where a fixed amount of GitHub repositories was displayed. At that point, we took the decision that users should be able to browse through a huge number of repositories that the GitHub Search API is exposing, so the only solution for that was to integrate paging within our app.

However, we then realized that we needed to first understand the concept of data streams in the context of pagination, so we learned a few things about Kotlin Flow and how it can be a simple solution to consume paginated content.

Then, we explored how the Jetpack Paging library is an elegant solution to adding pagination to our apps, culminating with the practical task of integrating paging in our Repositories App with the help of this library. Finally, we transformed our Repositories App into a modern application that creates the illusion of an infinite list of repositories, with initial and intermediary loading or error states, as well as retry functionality.

In the next chapter, we will tackle yet another Jetpack subject—Lifecycle components!

Further reading

In this chapter, we briefly covered how you can integrate Jetpack Paging into an Android application. However, in the context of pagination and

Jetpack Paging, there are a couple of more advanced topics that you might end up wondering about, as outlined here:

- **Having both a local and remote source for paginated content**—For such a case, you will need a component that manages communication between the two data sources. For this task, you could use the built-in **RemoteMediator** API of the Paging library. You can learn more about it from its official documentation at <https://developer.android.com/topic/libraries/architecture/paging/v3-network-db#implement-remotemediator>.
- **Adding support for content refresh or invalidation**—If you're looking to support pull-to-refresh functionality, or you're interested in making sure that the user is returned to the appropriate page upon various system events that could restart the paginated content, you need to obtain the refresh keys of the **PagingSource** component. Learn more about this from the official documentation at <https://developer.android.com/topic/libraries/architecture/paging/v3-migration#refresh-keys>.

As you know by now, testing is very important. In the context of paging, testing can get a little trickier. If you're interested in learning how to test your paging app, check out the official documentation at <https://developer.android.com/topic/libraries/architecture/paging/test>.

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

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