8

# Getting Started with Paging

In Android development, the Paging library helps developers load and display data pages from a larger dataset from local storage or over a network. This can be a common case if your application loads considerable amounts of data for people to read. For instance, a good example is Twitter; you might notice the data refreshes due to the many tweets that people send daily.

Hence, in **Modern Android Development** (**MAD**), Android developers might want to implement the Paging library in their applications to help them with such instances when loading data. In this chapter, you will learn how to utilize the Paging library in your projects.

In this chapter, we'll cover the following recipes:

- Implementing the Jetpack Paging library
- Managing present and loading states
- Implementing your custom pagination in Jetpack Compose
- Loading and displaying paged data
- Understanding how to transform data streams
- Migrating to Paging 3 and understanding the benefits
- Writing tests for your Paging Source

## Technical requirements

The complete source code for this chapter can be found at **https://github.com/PacktPublishing/Modern-Android-13-Development-Cookbook/tree/main/chapter_eight**. You will also need to get an API key for **https://newsapi.org/**. `NewsApi` is a worldwide API for news.

# Implementing the Jetpack Paging library

The Paging library comes with incredible features for developers. If your codebase is established and extensive, there are other custom ways that developers have created to help them load data efficiently. One notable advantage of Paging is its in-memory caching for your page's data, which ensures your application uses the system resources efficiently while working with the already paged data.

In addition, it offers support for Kotlin coroutine flows and LiveData and has built-in deduplication, which ensures your application uses network bandwidth and resources efficiently, which can help save battery. Finally, the Paging library offers support for error handling, including when refreshing and retrying your data.

## Getting ready

In this recipe, we will need to create a new project; if you need to reference a previous recipe for creating a new project, you can visit *Chapter 1, Getting Started with Modern Android Development Skills*.

## How to do it...

Let's go ahead and create a new empty Compose project and call it `PagingJetpackExample`. In our ex-

ample project, we will use the free **`NewsApi`** to display the news to our users. To get started, check out this link at **https://newsapi.org/docs/get-started**. Also, ensure you get your API for the project, as it is a requirement for this recipe. Follow these steps to get started:

1. Let's go ahead and add the following required dependencies. In addition, since we will be doing a network call, we need to add a library to handle this. As for the correct versioning, check out the *Technical requirements* section for the code and the correct version. We will provide **`2.x.x`** so you can check compatibility if you are upgrading or already have **`Retrofit`** in your project and Coil which is a fast, lightweight, and flexible image loading library. It is designed to simplify the process of loading images from various sources (such as network, local storage, or content providers) and displaying them in ImageView or other image-related UI components:

```
//Retrofit
implementation 'com.squareup.retrofit2:retrofit:2.x.x'
implementation 'com.squareup.retrofit2:converter-gson:2.x.x'
//Coil you can also use Glide in this case
implementation 'com.google.accompanist:accompanist-coil:0.x.x'
//Paging 3.0
implementation 'Androidx.Paging:Paging-compose:1.x.x'
```

2. After the project syncs and is ready, go ahead and remove the **`Greeting`** composable function that comes with the project. You should have just your theme, and your surface should be empty. In addition, for the **user interface** (**UI**) portion of this recipe, you can get the entire code from the *Technical requirements* section.
3. Also, when using an API, developers tend to forget to add the **`Android.permission.INTERNET`** permission on the manifest, so let's do that now before we forget it:

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

4. Now, create a package and call it **`data`**; we will add our model and service files to this package. In addition, ensure you go through the News API **Documentation** section to understand how the API works:

```
data class NewsArticle(
    val author: String,
    val content: String,val title: String ...)
```

5. Let us now create our **`NewsArticleResponse`** data class, which we will implement in our **`NewsApiService`** interface. Our API call type is **`@GET()`**, which means exactly "to get." A more detailed explanation of **`GET`** is provided in the *How it works* section. Our call seeks to return a call object containing the data in the form of the **`NewsArticleResponse`** data class:

```
data class NewsArticleResponse(
    val articles: List<NewsArticle>,
    val status: String,
    val totalResults: Int
)
interface NewsApiService{
    @GET("everything?q=apple&sortBy=popularity&apiKey=        ${YOURAPIKEY}&pageSize=20")
    suspend fun getNews(
        @Query("page") page: Int
    ): NewsArticleResponse
}
```

6. Create another class called **`NewsArticlePagingSource()`**; our class will use **`NewsApiService`** as the input parameter. When exposing any large datasets through APIs, we need to provide a mechanism to paginate the list of resources. To implement it, we need to pass the type of the Paging key and the type of data to load, which in our case is **`NewsArticle`**:

```
class NewsArticlePagingSource(
    private val newsApiService: NewsApiService,
): PagingSource<Int, NewsArticle>() {
```

```
. . .
}
```

7. Finally, let us go ahead and override **`getRefreshKey()`** provided by the **`PagingSource`** and **`load()`** suspend functions. We will discuss the **`load()`** and **`PagingSource`** suspend functions in detail in the *Loading and displaying paged data* recipe:

```
class NewsArticlePagingSource(
    private val newsApiService: NewsApiService,
) : PagingSource<Int, NewsArticle>() {
    override fun getRefreshKey(state: PagingState<Int,
    NewsArticle>): Int? {
        return state.anchorPosition?.let {
        anchorPosition ->
            state.closestPageToPosition(
                anchorPosition)?.prevKey?.plus(1)
                ?: state.closestPageToPosition(
                    anchorPosition)?.nextKey?.minus(1)
        }
    }
    override suspend fun load(params:
    LoadParams<Int>): LoadResult<Int, NewsArticle> {
        return try {
            val page = params.key ?: 1
            val response = newsApiService.getNews(
                page = page)
            LoadResult.Page(
                data = response.articles,
                prevKey = if (page == 1) null else
                    page.minus(1),
                nextKey = if
                    (response.articles.isEmpty()) null
                        else page.plus(1),
            )
        } catch (e: Exception) {
            LoadResult.Error(e)
        }
    }
}
```

8. Now, let's create our repository; a repository is a class that isolates the data sources, such as a web service or a Room database, from the rest of the

app. Since we do not have a Room database, we will work with the web service data:

```
class NewsArticleRepository @Inject constructor(
    private val newsApiService: NewsApiService
) {
    fun getNewsArticle() = Pager(
        config = PagingConfig(
            pageSize = 20,
        ),
        PagingSourceFactory = {
            NewsArticlePagingSource(newsApiService)
        }
    ).flow
}
```

9. We will use Hilt for Dependency Injection in our project and build the required modules that will be supplied. For this section, you can reference the steps in *Chapter 3*, *Handling the UI State in Jetpack Compose and Using Hilt*, on how to add Hilt to your project and also how to create the required modules. In addition, you can access the entire code through the *Technical requirements* section if you get stuck:

```
@Module
@InstallIn(SingletonComponent::class)
class RetrofitModule{
    @Singleton
    @Provides
    fun provideRetrofitInstance(): NewsApiService =
    Retrofit.Builder()
        .baseUrl(BASE_URL)
        .addConverterFactory(
            GsonConverterFactory.create())
        .build()
        .create(NewsApiService::class.java)
}
```

10. Finally, after we have implemented our **`PagingSource`**, we can go ahead and create a **`Pager`** which typically refers to a **`ViewPager`** in our ViewModel and specify our page size. This can range based on the project's needs or preferences. Furthermore, when using Paging 3.0, we

don't need to individually handle or convert any data to survive the screen configuration changes because this is done for us automatically.

We can simply cache our API result using **`cachedIn(viewModelScope)`**. In addition, to notify of any change to the **`PagingData`**, you can handle the loading state using a **`CombinedLoadState`** callback:

```
@HiltViewModel
class NewsViewModel @Inject constructor(
    private val repository: NewsArticleRepository,
) : ViewModel() {
    fun getNewsArticle():
        Flow<PagingData<NewsArticle>> =
            repository.getNewsArticle().cachedIn(
                viewModelScope)
}
```

11. Finally, when you run the application, you should see a display like *Figure 8.1*, showing the author's name, image, and content. We also wrap the content since this example is just for learning purposes; you can take it as a challenge to improve the UI and display more details:

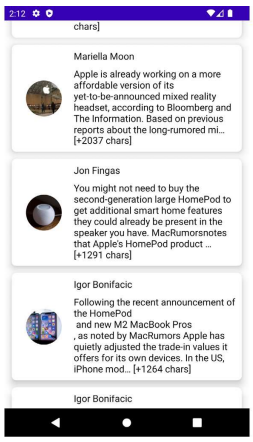


Figure 8.1 – The news article being loaded using the Paging 3 library

## How it works...

In Android development, a retrofit request typically refers to a network request made using the Retrofit library, a popular HTTP client library for Android.

Here are some common types of Retrofit requests and their usage:

- **`GET`**: This request is used to retrieve data from a server. It is the most common type of request used in Android apps and is often used to retrieve data to populate a UI element such as a list or a grid.
- **`POST`**: This request is used to submit data to a server. It is commonly used to create new resources on the server, such as a new user account or a new post.
- **`PUT`**: This request is used to update an existing resource on the server. It is commonly used to update a user's account information or to modify an existing post.
- **`DELETE`**: This request is used to delete a resource on the server. It is commonly used to delete a user account or to remove a post.
- **`PATCH`**: This request partially updates an existing resource on the server. It is commonly used when only a small portion of the resource needs to be updated rather than updating the entire resource with a **`PUT`** request.

When making Retrofit requests, developers typically define an interface that describes the endpoint and the request parameters. Retrofit then generates a client implementation for that interface, which can be used to make the actual network calls.

By using Retrofit, developers can abstract away many of the low-level details of network requests, making it easier and more efficient to communicate with a server from an Android app. For examples

about Retrofit, check out the following link **https://square.github.io/retrofit/**.

The Paging library ensures it adheres to the recommended Android architecture patterns. Furthermore, its components are the **Repository**, **ViewModel**, and **UI** layers. The following diagram shows how Paging components operate at each layer and how they work together in unison to load and display your paged data:

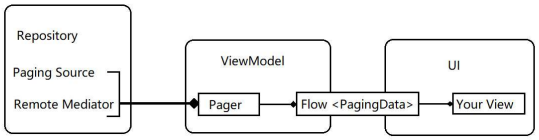


Figure 8.2 – The Paging library architecture

The **Paging Source** component is the main component in the **Repository** layer, as seen in *Figure 8.2*. The object usually declares a source for each piece of data and also handles how to retry data from that source. If you noticed, that is precisely what we did in our example:

```
    class NewsArticleRepository @Inject constructor(
        private val newsApiService: NewsApiService
    ) { . . .
```

We create our Retrofit **builder()** object that contains our base URL of the API, which we defined in the **Constant** class, **const val BASE_URL = "https://newsapi.org/v2/"**, and we use the **Gson** converter to convert our JSON API response. We then declare the **apiService** variable that we will use to connect the Retrofit **builder()** object with our interface and complete our retrofit module.

*IMPORTANT NOTE*

*It is recommended for anyone using the Paging Library to migrate to Paging 3 due to its improve-*

*ments and because some functionalities are hard to handle using Paging 2.*

# Managing present and loading states

The Paging library offers the loading state information to users through its load state object, which can have different forms based on its current loading state. For example, if you have an active load, then the state will be **`LoadState.Loading`**.

If you have an error state, then the state will be a **`LoadState.Error`**; and finally, there might be no active load operation, and this state is called the **`LoadState.NotLoading.`** In this recipe, we will explore the different states and get to understand them; the example demonstrated here can also be found at the following link: **https://developer.android.com/topic/libraries/architecture/paging/load-state**. In this example, we assume your project uses legacy code, which utilizes XML for the view system.

## Getting ready

To follow along with this recipe, you need to have completed the code in the previous recipe. You can also skip this if it is not required in your project.

## How to do it...

We will not create a new project in this recipe but rather a step-by-step look at how we can access the loading state with a listener or present the loading state with an adapter. Follow along with these steps to get started:

1. When you want to access the state, pass this information to your UI. You can easily use the **`loadedStateFlow`** stream of the **`addLoadStateListener`** function provided by **`PagingDataAdapter`**:

```
lifecycleScope.launch {
    thePagingAdapter.loadStateFlow.collectLatest {
        loadStates ->
    progressBar.isVisible = loadStates.refresh is
        LoadState.Loading
    retry.isVisible = loadState.refresh !is
        LoadState.Loading
    errorMessage.isVisible = loadState.refresh is
        LoadState.Error
  }
}
```

2. For our example, we will not look into the **`addLoadStateListener`** function since this is used with an adapter class, and with the new Jetpack Compose, this is barely performed since there is more of a push to use the Jetpack Compose UI-based applications.
3. Filtering the load state steam might make sense based on your application's specific event. This ensures that your app UI is updated at the correct time to avoid issues. Hence, using coroutines, we wait until our refresh load state is updated:

```
lifecycleScope.launchWhenCreated{
    yourAdapter.loadStateFlow
        .distinctUntilChangedBy { it.refresh }
        .filter { it.refresh is LoadState.NotLoading }
        .collect { binding.list.scrollToPosition(0) }
}
```

## How it works...

When getting updates from **`loadStateFlow`** and **`addLoadStateListener()`**, these are guaranteed to be synchronous, and they update the UI as needed. This simply means in the Paging 3 library for Android, **`LoadState.Error`** is a state that indicates

an error has occurred while loading data from a `PagingSource`.

In Paging 3 library for Android, `LoadState.NotLoading` is a state that indicates that the `PagingDataAdapter` is not currently loading any data and that all available data has been loaded.

When a `PagingDataAdapter` is first created, it starts in the `LoadState.NotLoading` state. This means that no data has been loaded yet, and the adapter is waiting for the first load to occur.

After the first load, the adapter may transition to a different load state depending on the current state of the data loading process. However, once all available data has been loaded, the adapter will transition back to the `LoadState.NotLoading` state.

`LoadState.NotLoading` can be used to inform the UI that the data-loading process is complete and that no further data will be loaded unless the user initiates a refresh or other action.

To handle this state, you can register a listener for changes to the `LoadState` in the `PagingDataAdapter` and update the UI accordingly. For example, you could display a message to the user indicating that all data has been loaded or disable any “load more” buttons or gestures.

## There’s more…

You can learn more about the state and how to better handle Paging by following this link: **https://developer.android.com/topic/libraries/architecture/paging/load-state**.

# Implementing your custom pagination in Jetpack Compose

The Paging library has incredible features for developers, but sometimes you encounter challenges and are forced to create custom pagination. At the beginning of the chapter, we talked about complex code bases having or creating pagination.

In this recipe, we will look into how we can achieve this with a simple list example and how you can use this example to create custom pagination in your application.

## Getting ready

In this recipe, we will need to create a new project and call it **`CustomPagingExample`**.

## How to do it...

In our example project, we will try to create a student profile card and use custom pagination to load the profiles in Jetpack Compose.

1. For this recipe, let us go ahead and add the **`lifecycle-ViewModel`** dependency since we will need it:

```
implementation "Androidx.lifecycle:lifecycle-viewmodel-compose:2.x.x"
```

2. Let's go ahead and create a new package and call it **`data`**. In our **`data`** package, we will add the items we will display on our card. For now, we will just display the student's **`name, school, and major`**:

```
data class StudentProfile(
    val name: String,
    val school: String,
```

```
    val major: String
)
```

3. Now that we have our **`data`** class, we will go ahead and build our repository, and since, in our example, we are not using an API, we will use our remote data source, and we can try to load, say, 50 to 100 profiles. Then, inside **`data`**, add another class and call it **`StudentRepository`**:

```
class StudentRepository {
    private val ourDataSource = (1..100).map {
        StudentProfile(
            name = "Student $it",
            school = "MIT $it",
            major = "Computer Science $it"
        )
    }
    suspend fun getStudents(page: Int, pageSize: Int):
    Result<List<StudentProfile>> {
        delay(timeMillis = 2000L) //the delay added is
            just to mimic a network connection.
        val start = page * pageSize
        return if (start + pageSize <=
        ourDataSource.size) {
            Result.success(
                ourDataSource.slice(start until start
                    + pageSize)
            )
        } else Result.success(emptyList())
    }
}
```

4. Now that we have created our repository let us go ahead and create our custom pagination. We will do this by creating a new interface and calling it **`StudentPaginator`**:

```
interface StudentPaginator<Key, Student> {
    suspend fun loadNextStudent()
    fun reset()
}
```

5. Since **`StudentPaginator`** is an interface, we must create a class to implement the two functions we just created. Now, let us go ahead and create

**StudentPaginatorImpl** and implement our interface:

```
class StudentPaginatorImpl<Key, Student>(
) : StudentPaginator<Key, Student> {
    override suspend fun loadNextStudent() {
        TODO("Not yet implemented")
    }
    override fun reset() {
        TODO("Not yet implemented")
    }
}
```

6. Next, you will need to work on what you need to handle in the **StudentPaginator** implementation class. For instance, in our constructor, we will need to create a key to listen to the **load**, **request**, **error**, **success**, and **next key**, and then finally, on the **reset()** function, be able to reset our pagination. You can view the complete code in the *Technical requirements* section. You might also notice it looks similar to the Paging Source in the first recipe of this chapter:

```
class StudentPaginatorImpl<Key, Student>(
    private val key: Key,
    private inline val loadUpdated: (Boolean) -> Unit,
    private inline val request: suspend (nextKey: Key)
    ->
. . .
) : StudentPaginator<Key, Student> {
    private var currentKey = key
    private var stateRequesting = false
    override suspend fun loadNextStudent() {
        if (stateRequesting) {
            return
        }
        stateRequesting = true
     . . .
    }
    override fun reset() {
        currentKey = key
    }
```

7. Let's go ahead and create a new package and call it **uistate**. Inside **uistate**, we will create a new

data class and call it **`UIState`** to help us handle the UI state:

```
data class UIState(
    val page: Int = 0,
    val loading: Boolean = false,
    val studentProfile: List<StudentProfile> =
        emptyList(),
    val error: String? = null,
    val end: Boolean = false,
)
```

8. Now let's go ahead and finalize our **`ViewModel`**

   **`init`** in Kotlin is the block that we use for our initialization. We also create **`val ourPaginator`** that we declare to the **`StudentPaginatorImpl`** class and handle the inputs with the data we need for our UI:

```
class StudentViewModel() : ViewModel() {
    var state by mutableStateOf(UIState())
    private val studentRepository =
        StudentRepository()
    init {
        loadStudentProfile()
    }
    private val ourPaginator = StudentPaginatorImpl(
        key = state.page,
        loadUpdated = { state = state.copy(loading =
            it) },
        request = { studentRepository.getStudents(it,
            24) },
        nextKey = { state.page + 1 },
        error = { state = state.copy(error =
            it?.localizedMessage) },
        success = { student, newKey ->
            state = state.copy(
                studentProfile = state.studentProfile
                    + student,
                page = newKey,
                end = student.isEmpty()
            )
        }
    )
    fun loadStudentProfile(){
        viewModelScope.launch {
            ourPaginator.loadNextStudent()
```

```
            }
        }
    }
```

9. Finally, in our **`MainActivity`** class, we now load the student profile on our card and display it on the screen, as shown in *Figure 8.3*. A tremendous additional exercise to try out is to use Dependency Injection on the sample project to enhance your Android skills. You can utilize *Chapter 3*, *Handling the UI State in Jetpack Compose and Using Hilt*, for adding Dependency Injection and also to try writing tests for the **`ViewModel`** class:

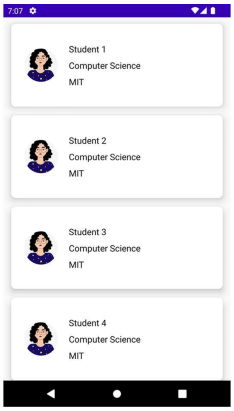


Figure 8.3 – The data loaded on a lazy column

In *Figure 8.4* you will see a progress loading symbol when you scroll down to **Student 4** and so on, which can be great when you have huge loads of data:

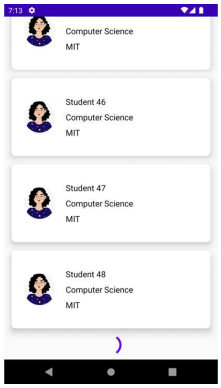


Figure 8.4 – Our data being loaded

## How it works...

You might experience issues once you get a list, and it might be tough to notify single items. However, you can easily make your pagination; in our project, we simulate a remote data source but remember that you can use any API for this example.

Our primary focus is the **`StudentPaginatorImpl`** class – you will notice we pass in a key, a **`loadUpdated`** value, and a request which is a suspend function that returns a result from our **`Student`** type; we also pass the **`nextkey`**, which tells us where we are. Then, in case of an error, we have the throwable error and a **`suspend`** value, **`success`**, which gives us the **`success`** result:

```
class StudentPaginatorImpl<Key, Student>(
    private val key: Key,
    private inline val loadUpdated: (Boolean) -> Unit,
    private inline val request: suspend (nextKey: Key) ->
        Result<List<Student>>,
    private inline val nextKey: suspend (List<Student>) ->
        Key,
    private inline val error: suspend (Throwable?) -> Unit,
    private inline val success: suspend (items:
        List<Student>, newKey: Key) -> Unit
) : StudentPaginator<Key, Student> {
```

So when we override our function from the **`loadNextStudent()`** interface, we first check our current state request and return our initial value as **`false`**, but we update it after our status check. We also ensure that we reset the key by setting the **`currentKey`** to the **`nextKey`**.

```
currentKey = nextKey(studentProfiles)
success(studentProfiles, currentKey)
loadUpdated(false)
```

This makes it easy if you ever need to customize an item in your **`LazyColumn`**, ensuring you have great lists.

The `loadStudentProfile()` function has a `viewModelScope.launch {...}`. A ViewModel scope is defined for each ViewModel in our application. In addition, any coroutine launched in this scope is auto-canceled if the ViewModel is cleared.

You might be wondering what a ViewModel is. To help refresh your knowledge, you can look into *Chapter 3*, *Handling the UI State in Jetpack Compose and Using Hilt*.

# Loading and displaying paged data

There are essential steps to consider when loading and displaying paged data. In addition, the Paging library provides tremendous advantages for loading and displaying large, paged datasets. A few steps you must have in mind is ensuring you first define a data source, your Paging Source set up streams if needed, and more.

In this recipe, we will look at how loading and displaying paged data works.

## How to do it...

You need to have completed the *Implementing the Jetpack Paging library* recipe to be able to follow along with the explanation of this recipe:

1. You might have noticed in our first recipe that we override `load()`, a method that we use to indicate how we retrieve the paged data from our corresponding data source:

```
override suspend fun load(params: LoadParams<Int>): LoadResult<Int, NewsArticle> {
    return try {
        val page = params.key ?: 1
        val response = newsApiService.getNews(page =
```

```
                page)
            LoadResult.Page(
                data = response.articles,
                prevKey = if (page == 1) null else
                    page.minus(1),
                nextKey = if (response.articles.isEmpty())
                    null else page.plus(1),
            )
        } catch (e: Exception) {
            LoadResult.Error(e)
        }
    }
```

2. We start refreshing at page **`1`** if **`val page = params.key ?: 1`** is undefined when we override **`getRefreshKey()`**; we try to find the page key of the closest page to the anchor position from either our previous key or the next key. We also need to ensure we handle cases where we might have some **`null`** values:

```
override fun getRefreshKey(state: PagingState<Int, NewsArticle>): Int? {
    return state.anchorPosition?.let { anchorPosition
        ->
        state.closestPageToPosition(anchorPosition)?
            .prevKey?.plus(1)
            ?: state.closestPageToPosition(
                anchorPosition)?.nextKey?.minus(1)
    }
}
```

## How it works...

When using the Paging library, you can specify the position of the first item to be displayed on the screen using the **`anchorPosition`** parameter. In addition, **`anchorPosition`** is an optional parameter that you can pass to the **`PagingItems`** composable function, which is used to display paged data. The **`anchorPosition`** parameter is used to specify the position of the first item to be displayed on the screen when the composable is first rendered.

The `LoadParams` object carries the information about the load operation to be performed. In addition, it knows about the key to be loaded and the number of items to be displayed on the UI. Furthermore, to better understand how the `load()` function receives the key for each specific load and updates it, review the following diagram:

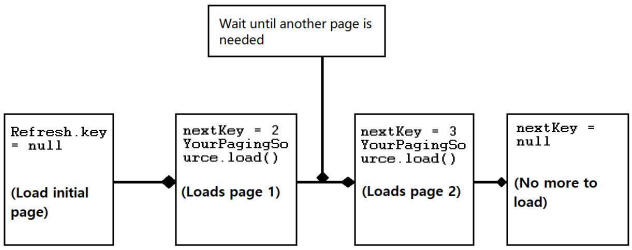


Figure 8.5 – How load() uses and updates the key

# Understanding how to transform data streams

When writing any code dealing with Paging, you need to understand how you can transform the data stream as you load it to your users. For instance, you may need to filter a list of items or even convert the items to a different type before you can feed the UI with the data.

Hence, ensuring you apply transformation directly to the stream data lets you keep your repository and UI logic separated cleanly. In this recipe, we will try to understand how we can transform data streams.

## Getting ready

To follow along, you must be familiar with the primary usage of the Paging library; hence make sure you have read the previous recipes in this chapter.

## How to do it...

In this recipe, we will perform the following steps:

1. Look into how we can apply the essential transformation.
2. Convert and filter the data.
3. Handle separators in the UI and convert the UI model.

The recipe is helpful to you if you are already using Paging in your application.

4. First, we need to place the transformation inside a **`map{PagingData ->}`**. A map in Kotlin applies the given lambda function to each element and returns a list of the lambda results:

```
yourPager.flow
    .map { PagingData ->
        // here is where the transformations are
           applied to the items in the paged data.
}
```

5. Second, when we want to convert the data or filter, once we have access to our **`PagingData`** object, we can use **`map()`** again on each item separately in the paged list. A typical use case is when you want to map a database or network layer object onto an object that might be used in the UI layer specifically:

```
yourPager.flow
    .map { PagingData ->
        PagingData.map { sports -> SportsModel(sports)
        }
}
```

6. We will need to place the filter operation inside the map because the filter applies to the **`PagingData`** object. Then once the data is filtered out from our **`PagingData`**, the new instance is paged to the UI layer and displayed:

```
yourPager.flow
    .map { PagingData ->
        PagingData.filter { sports ->
```

```
                !sports.displayInUi }
    }
```

7. Finally, when handling separators in the UI or converting the UI model, the most significant steps are ensuring that you do the following:
    1. Convert the UI models to accommodate your separator items.
    2. Transform the data dynamically and add the separators between presenting and loading the data.
    3. Update the UI to handle the separator items better.

## How it works...

The **`PagingData`** is encapsulated in a reactive stream; what this means is that before loading the data and displaying it to the users, you can incrementally apply the transform to the data. Transforming data streams can be crucial when you have a complex application, and handling this situation in advance might help ensure your application scales better and help minimize the complexity of your data growth.

## See also

It is fair to acknowledge that this recipe cannot cover all the information you need to know about transforming the data stream. That said, if you encounter an issue and want to learn more, you can always reference the following link to learn more about how you can handle separators in the UI and more: **https://developer.android.com/topic/libraries/architecture/paging/v3-transform**.

# Migrating to Paging 3 and understanding the

# benefits

You might be using the old Paging version, in this case, Paging 2 or 1, and you might be required to migrate to utilize the benefits Paging 3 offers. Paging 3 offers enhanced functionality and ensures it addresses the most common challenges people experience using Paging 2.

In this recipe, we will look into how you can migrate to the latest recommended Paging library.

## Getting ready

If your application is already using Paging 3, then you can skip this recipe; this step-by-step migration guide is intended for users currently using the older versions of the Paging library.

## How to do it...

Migrating from old versions of the Paging library might seem complex due to the fact that each application is unique, and complexities might vary. In our example, however, we will touch on a low-level kind of migration since our example application does not need any migration.

To perform migration from old Paging libraries, follow these steps:

1. The first step is to replace the refresh keys, and this is because we need to define how refreshing resumes from the middle of loading data. We will do this by first implementing **`getRefreshKey()`**, which maps the correct initial key using **`PagingState.anchorPosition`** as the recent index:

```
override fun getRefreshKey(PagingState: PagingState): String? {
    return PagingState.anchorPosition?.let { position
```

```
            ->
                PagingState.getClosestItemToPosition(
                    position)?.id
        }
    }
```

2. Next, we need to ensure we replace the positional data source:

```
override fun getRefreshKey(PagingState: PagingState): Int? {
    return PagingState.anchorPosition
}
```

3. If you are using the old Paging library, the paged data uses **`DataSource.map()`**, **`mapByPage`**, **`Factory.map()`**, and **`Factory.mapByPage`**. In Paging 3, however, all these are applied as operators to **`PagingData`**.
4. Finally, to ensure you migrate from **`PageList`**, which is in Paging 2, you will need to migrate to **`PagingData`**. The most notable change is that **`PagedList.Config`** is not **`PagingConfig`**. In addition, the **`Pager()`** exposes an observable **`Flow<PagingData>`** with its flow:

```
val yourFlow = Pager(
    PagingConfig(pageSize = 24)
) {
    YourPagingSource(yourBackend, yourQuery)
}.flow
    .cachedIn(viewModelScope)
```

## How it works...

To ensure your migration is complete and successful, you must make sure you migrate all the significant components from Paging 2. This includes the **`DataSource`** classes, **`PagedList`**, and **`PagedListAdapter`** if your application uses it. Furthermore, some Paging 3 components work well with other versions, which simply means it is backward compatible.

The most notable change to **PagingSource** in Paging 3 is that it combines all the loading functions into one, now called **load()** in **PagingSource**. This ensures there is no redundancy in the code because the loading logic is often identical to the old API. In addition, the loading function parameters in Paging 3 now use the **LoadParams** sealed class, which has subclasses for each load type.

In **PagedList**, which is used in Paging 2, when you migrate, you might use **PagingData** and **Pager**. When you start to use **PagingData** from Paging 3, you should ensure that the configuration is moved from the old **PagedList.Config** to **PagingConfig**.

# Writing tests for your Paging Source

Writing tests for your implementations is crucial. We will write unit tests for our **PagingSource** implementation in this recipe to test our logic. Some tests that might be worth writing are checking when news Paging load failure happens.

We can also test the success state and more. You can follow the pattern to write tests for your project or use case.

## Getting ready

To follow this recipe step by step, you need to have followed the *Implementing the Jetpack Paging library* recipe, and you need to use the **PagingJetpackExample** project.

## How to do it...

Open **PagingJetpackExample** and follow along with this project to add unit tests:

1. Add the following testing libraries to your **build.gradle** app:

```
testImplementation 'org.assertj:assertj-core:3.x.x'
testImplementation "org.mockito:mockito-core:3.x.x"
testImplementation 'Androidx.arch.core:core-testing:2.x.x'
testImplementation 'org.jetbrains.kotlinx:kotlinx-coroutines-test:1.x.x'
```

2. After adding the dependencies, create a new package and call it **data** in your **test** package in the project structure. You can reference the *Understanding the Android project structure* recipe in *Chapter 1*, *Getting Started with Modern Android Development Skills*, if you need help finding the folder.
3. Create a test class and call it **NewsArticlePagingSourceTest**.
4. Inside the class, let's go ahead and add **Mock** to mock our **ApiService** interface and create a **lateinit var newsApiService** that we will initialize at our **@Before** step:

```
@Mock
private lateinit var newsApiService: NewsApiService
lateinit var newsPagingSource: NewsArticlePagingSource
```

5. Now let's go ahead and create our **@Before** so we can run our **CoroutineDispatchers**, which is used by all standard builders such as async, and launch to our **@Before** step too:

```
@Before
fun setup() {
    Dispatchers.setMain(testDispatcher)
    newsPagingSource =
        NewsArticlePagingSource(newsApiService)
}
```

6. The first test we will need to write is to check when a failure happens. Hence let's go ahead and set up our test. A **403** response is a forbidden status code indicating the server understood your request but did not authorize it:

```
@Test
fun `news article Paging Source load failure http error`() = runBlockingTest {
```

```
        //setup
        val error = HttpException(
            Response.error<ResponseBody>(
                403, "some content".toResponseBody(
                    "plain/text".toMediaTypeOrNull())
            )
        ) . . .
```

7. To continue our test, we will need to use **`Mockito.doThrow(error)`**:

```
Mockito.doThrow(error)
    .`when`(newsApiService)
    .getNews(
        1
    ). . .
```

8. Then, finally, we trigger **`PagingSource.LoadResult.Error`** and pass in the type, then assert:

```
//assert
assertEquals(
    expectedResult, newsPagingSource.load(
        PagingSource.LoadParams.Refresh(
            key = null,
            loadSize = 1,
            placeholdersEnabled = false
        )
    )
)
```

9. You can add two more additional tests and then add **`tearDown`** to clean up the coroutines:

```
@After
fun tearDown() {
    testDispatcher.cleanupTestCoroutines()
}
```

## How it works...

We use **`Mock`** in unit tests, and the general idea is based on the notion that the objects under tests might have dependencies on other complex objects. Based on this, it is much easier to isolate the behavior of the object we want by mocking the object,

which ensures it has the same behavior as our real object and makes testing easier:

```
@Mock
private lateinit var newsApiService: NewsApiService
```

Our **`lateinit var newsPagingSource: NewsArticlePagingSource`** is used for late initialization, and we initialize it on our **`@Before`** function.