7

# Getting Started with WorkManager

In Android, `WorkManager` is an API introduced by Google as part of the Android Jetpack library. It is a powerful and flexible background task scheduling library that allows you to perform deferrable, asynchronous tasks even when your app is not running or the device is in a low-power state.

`WorkManager` provides a unified API to schedule tasks that need to be executed at a specific time or under certain conditions. It takes care of managing and running tasks efficiently, depending on factors such as device idle state, network connectivity, and battery level.

Furthermore, `WorkManager` allows observation of work status and chain creation. This chapter will look into how we can implement `WorkManager` using examples and learn how it works and its use cases.

In this chapter, we'll be covering the following recipes:

- Understanding the Jetpack `WorkManager` library
- Understanding `WorkManager` state
- Understanding threading in `WorkManager`
- Understanding chaining and canceling work requests
- Implementing migration from Firebase `JobDispatcher` to the new recommended `WorkManager`
- How to debug `WorkManager`
- Testing `Worker` implementations

## Technical requirements

This chapter utilizes step-by-step examples and does not create a complete project. `WorkManager` is helpful, but because the

use case may vary, utilizing examples to see how the code fits your need is an excellent art to learn in programming.

# Understanding the Jetpack WorkManager library

**WorkManager** is one of the most powerful Jetpack libraries, and it is used for persistent work. The API allows observation of persistent status and the ability to create a complex chain of work. When building Android applications, it might be a requirement to have your data persist; if you need help to refresh your knowledge, you can reference *Chapter 6*, *Using the Room Database and Testing*.

`WorkManager` is the most-recommended API for any background process and is known to handle unique types of ongoing work as shown here:

- **Immediate**: As the name suggests, these are tasks that must be done immediately or completed soon
- **Long-Running**: Tasks that run for a long time
- **Deferrable**: A task that can be rescheduled and can be assigned a different start time and can also run periodically

Some more sample use cases where you can use `WorkManager` are, for instance, if your company wants to create custom notifications, send analytics events, upload images, periodically sync your local data with the network, and more. Furthermore, `WorkManager` is the favored API and is highly recommended as it replaces all previous background scheduling APIs in Android.

There are other APIs that are used for scheduling work. They are deprecated, and in this book, we will not cover them but will mention them since you might encounter them in work with legacy code; they are as follows:

- Firebase Job Dispatcher
- Job Scheduler
- GCM `NetWorkManager`
- `WorkManager`

## Getting ready

In this recipe, we will go ahead and look at a simple example of how we can create our own custom notification using `WorkManager`.

You can also use the same concept to send logs or report analytics for your application if you are listening to any logs. We opt for this task because sending notifications to your users is crucial, and most applications do this, compared to uploading images. In addition, with Android 13 and the new API, it's mandatory to request `android.permission.POST_NOTIFICATIONS`.

## How to do it...

For this recipe, you do not need to create a project, as the concepts can be used in an already-built project; instead, we will look at examples and walk through the examples with explanations:

1. We will need to ensure we have the required dependency:

```
implementation "androidx.work:work-runtime-ktx:
version-number"
```

You can get the latest **version number** by following the documentation at **https://developer.android.com/jetpack/androidx/releases/work**.

2. Let us now go ahead and create our notification channel. For this, Google offers a great guide on how you can create one at **https://developer.android.com/develop/ui/views/notifications/channels**, so copy the following code:

```
private fun createCustomNotificationChannel() {
    if (Build.VERSION.SDK_INT >=
    Build.VERSION_CODES.O) {
        val name = getString(
            R.string.notification_channel)
        val notificationDescription = getString(
            R.string.notification_description)
        val importance =
            NotificationManager.IMPORTANCE_DEFAULT
        val channel = NotificationChannel(CHANNEL_ID,
        name, importance).apply {
            description = notificationDescription
        }
```

```
            // Register the channel with the system
            val notificationManager: NotificationManager =
                getSystemService(
                    Context.NOTIFICATION_SERVICE) as
                        NotificationManager
            notificationManager.createNotificationChannel(
                channel)
        }
    }
```

Also, note that creating different channels for separating notification types is possible. As recommended in Android 13, this makes it easier for users to turn them on and off if they do not need them. For example, a user might want to be aware of the latest brands your app is selling, compared to you sending your users info about old existing brands.

3. Now we can create our **`workManagerInstance`**. Let us think of a scenario where we need to fetch data from our servers every 20 or 30 mins and check whether notifications are available. In that case, we might encounter an issue where users are no longer using our application, which means the application will be put in the background, or the process might even be killed.

Hence the question becomes how do we fetch the data when the application is killed? This is when **`WorkManager`** comes to the rescue.

4. We can now create an instance of **`WorkManager`**:

```
val workManagerInstance = WorkManager.getInstance(application.applicationContext)
```

5. We will now need to go ahead and set the constraints:

```
val ourConstraints = Constraints.Builder()
    .setRequiredNetworkType(NetworkType.CONNECTED)
    .setRequiresBatteryNotLow(false)
    .build()
```

6. We will also need to set data to pass to the worker; hence we will create new value data, then we will put a string to the endpoint request:

```
val data = Data.Builder()
data.putString(ENDPOINT_REQUEST, endPoint)
```

7. Now we can go ahead and create our **`PeriodicWorkRequestBuilder<GetDataWorker>`**. In our work, we will set the constraints, set our input data, and

pass the **`GetDataWorker()`** type, which we will create and then build. Furthermore, since we want to be fetching the data every 20 or 30 mins from our server, we use **`PeriodicWorkRequestBuilder<Type>()`** for that purpose:

```
val job =
    PeriodicWorkRequestBuilder<GetDataWorker>(20,
        TimeUnit.MINUTES)
    .setConstraints(ourConstraints)
    .setInputData(data.build())
    .build()
```

8. We can now finally call **`workManagerInstance`** and enqueue our job:

```
workManagerInstance
    .enqueue(work)
```

9. We can now go ahead and construct our **`GetDataWorker()`**. In this class, we will extend the **`Worker`** class, which will override the **`doWork()`** function. In our case, however, instead of extending the **`Worker`** class, we will extend the **`CoroutineWorker(context, workerParameters)`**, which will help in our case since we will collect this data in a flow. We will also be using Hilt, so we will call **`@HiltWorker`**:

```
@HiltWorker
class GetDataWorker @AssistedInject constructor(
    @Assisted context: Context,
    @Assisted workerParameters: WorkerParameters,
    private val viewModel: NotificationViewModel
) : CoroutineWorker(context, workerParameters) {
    override suspend fun doWork(): Result {
        val ourEndPoint = inputData.getString(
            NotificationConstants.ENDPOINT_REQUEST)
        if (endPoint != null) {
            getData(endPoint)
        }
        val dataToOutput = Data.Builder()
            .putString(
              NotificationConstants.NOTIFICATION_DATA,
              "Data")
            .build()
        return Result.success(dataToOutput)
    }
```

In our case, we are returning **`success.`** In our **`getData()`** function, we pass in the endpoint, and we can assume our data has two or three crucial attributes: the ID, the title, and the description.

10. We can now send notifications:

```
val notificationIntent = Intent(this, NotifyUser::class.java).apply {
    flags = Intent.FLAG_ACTIVITY_NEW_TASK or
        Intent.FLAG_ACTIVITY_CLEAR_TASK
}
notificationIntent.putExtra(NOTIFICATION_EXTRA, true)
notificationIntent.putExtra(NOTIFICATION_ID, notificationId)
val notifyPendingIntent = PendingIntent.getActivity(
    this, 0, notificationIntent,
    PendingIntent.FLAG_UPDATE_CURRENT
)
val builder = NotificationCompat
    .Builder(context, Channel_ID_DEFAULT)
    .setSmallIcon(notificationImage)
    .setContentTitle(notificationTitle)
    .setContentText(notificationContent)
    .setPriority(NotificationCompat.PRIORITY_HIGH)
    .setContentIntent(notifyPendingIntent)
    .setAutoCancel(true)
with(NotificationManagerCompat.from(context)) {
    notify(notificationId, builder.build())
}
```

11. We also need to create a **`PendingIntent.getActivity()`**, which means when there is a click on the notification, the user will start an activity. For this to happen, we can **`getStringExtra(NotificationConstants.NOTIFICATION_ID)`** when a notification is clicked and put extras in our intent. This will need to happen in our activity:

```
private fun verifyIntent(intent: Intent?) {
    intent?.let {
        if (it.hasExtra(
           NotificationConstants.NOTIFICATION_EXTRA)){
           it.getStringExtra(
           NotificationConstants.NOTIFICATION_ID)
        }
    }
}
```

12. And on our **`onResume()`**, we can now call our **`verifyIntent()`** function:

```
override fun onResume() {
    super.onResume()
    verifyIntent(intent)
}
```

And that's it; we have custom notifications using our **`WorkManager()`**.

## How it works...

When creating a notification, the `importance` parameter helps determine how to interrupt the user for any given channel, hence why one should specify it in the `NotificationChannel` constructor. If the importance is high and the device is running Android 5.0+, you're going to see a heads-up notification, otherwise, it will just be the icon in the status bar. However, it is essential to note that all notifications, regardless of their importance, appear in a non-interruptive UI at the top of your screen.

The `WorkManager` word is very straightforward, hence removing ambiguity from the API. When using `WorkManager`, `Work` is referenced utilizing the `Worker` class. In addition, the `doWork()` function that we call runs asynchronously in the background thread offered by the `WorkManager()`.

The `doWork()` function returns a `Result{}`, and this result can be `Success`, `Failure`, or `Retry`. When we return the successful `Result{}`, the work will be done and finished successfully. `Failure`, as the name suggests, means the work failed, and then we call `Retry`, which retries the work.

In our `GetDataWorker()`, we pass in `NotificationViewModel` and inject it into our worker using Hilt. Sometimes you might encounter a conflict. The good thing is there is support for such a case with four options for handling any conflict that might occur.

This case is unique to when you are scheduling unique work; it makes sense to tell `WorkManager` what action must be taken when a conflict arises. You can solve this problem easily by using the existing work policy, `ExisitingWorkPolicy`, which has `REPLACE`, `KEEP APPEND`, and `APPEND_OR_REPLACE`.

`Replace`, as the name suggests, replaces the existing work, while `Keep` keeps existing work and ignores new work. When you call *Append*, this adds the new work to the existing one, and finally, `Append or Replace` simply does not depend on the pre-requisite work state.

*IMPORTANT NOTE*

*`WorkManager` is a singleton, hence it can only be initialized once, that is, either in your app or in the library. And, if you are using any workers with custom dependencies, then you have to provide a `WorkerFactory()` to the config at the time of custom initialization.*

### There's more...

We can only cover some `WorkManager` steps here. Google has great sample code labs that you can follow through and understand how to use `WorkManager`.

To read more about `WorkManager`, you can use this link: **https://developer.android.com/guide/background/persistent**.

## Understanding WorkManager state

In the previous recipe, *Understanding the Jetpack WorkManager library*, we looked into how we can use `WorkManager`. In that recipe, you might have noticed `Work` goes through a series of state changes, and the `doWork` function returns a result.

In this recipe, we will explore states in depth.

### How to do it...

We will continue working on an example of how you can apply the concepts learned about in this recipe to your already-built project:

1. You might have noticed we mentioned before that we have three states: `Success`, `Failure`, and `Retry`. `Work` states, however, have different types of processes; we can have a one-time work state, periodic work state, or blocked state:

```
Result
    SUCCESS, FAILURE, RETRY
```

You can look into this abstract class in more depth by clicking on the result and seeing how it is written.

2. In the first recipe, *Understanding the Jetpack WorkManager library*, we looked into the steps of setting up **`WorkManager`**. Another great example is downloading files. You can override the **`fun doWork()`** and check whether your URI is not equal to null and return a success, else failure:

```
override suspend fun doWork(): Result {
    val file = inputData.getString(
        FileParameters.KEY_FILE_NAME) ?: ""
    if (file.isEmpty()){
        Result.failure()
    }
    val uri = getSavedFileUri(fileName = file,
        context = context)
    return if (uri != null){
        Result.success(workDataOf(
            FileParameters.KEY_FILE_URI to
                uri.toString()))
    }else{
        Result.failure()
    }
}
```

3. When handling a state, you can easily check when the state successfully specifies an action, when it failed to perform an action, and finally, when **`WorkInfo.State`** is equals to **`RUNNING`**, call **`running()`**; see the following code snippet:

```
when (state) {
    WorkInfo.State.SUCCEEDED -> {
        success(
            //do something
        )
    }
    WorkInfo.State.FAILED -> {
        failed("Downloading failed!")
    }
    WorkInfo.State.RUNNING -> {
        running()
    }
    else -> {
        failed("Something went wrong")
    }
}
```

4. The success result returns an instance of **`ListenableWorker.Result`**, used to indicate that the work was completed successfully.
5. For the mentioned states, you can either use **`enqueueUniqueWork()`**, which is used for one time, or **`PeriodicWorkRequestBuilder`**, which is used for periodic

work. In our example, we used `PeriodicWorkRequestBuilder<Type>`:

```
WorkManager.enqueueUniqueWork()
WorkManager.enqueueUniquePeriodicWork()
```

## How it works...

We always start our request with the *Enqueued* state for the one-time work state, which means the work will run as soon as the constraints are met. Thereafter, we move to *Running*, and if we hit a *Success*, the work is done.

If in any instance, we end up *Running* and we don't hit *Success*, then it means we failed. Then, we will move back to *Enqueued* since we need to retry. *Figure 7.1* and *Figure 7.2* explain the states better for both one-time work and periodic work states.

Finally, if it happens that our enqueued work gets cancelled, then we move it to cancelled.

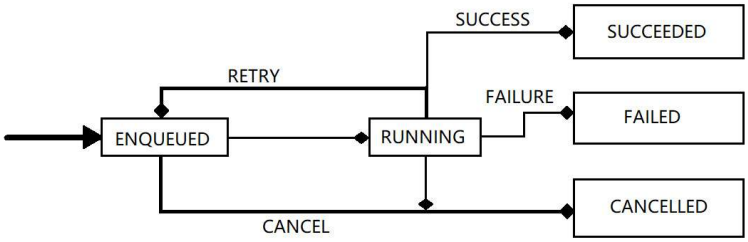


Figure 7.1 – How one-time work requests work

While the preceding image shows the one-time work state, the following diagram depicts the periodic work state.

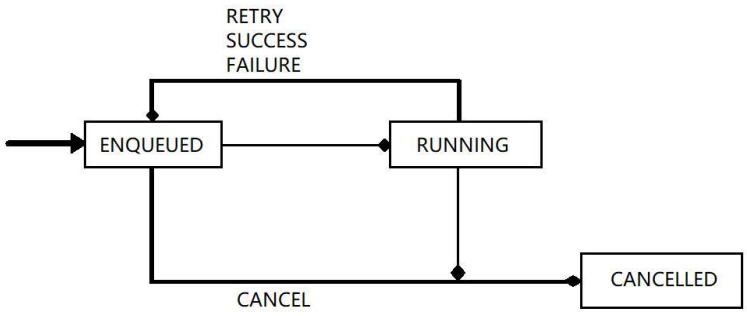


Figure 7.2 – How the periodic work state works

# Understanding threading in WorkManager

You can think of **`WorkManager`** as any process that runs in a background thread. When we use the **`Worker()`**, and **`WorkManager`** calls the **`doWork()`** function, this action works in the background thread. In detail, the background thread comes from the **`Executor`** specified in the **`WorkManager`** configuration.

You can also create your own custom executor for your application needs, but if that's not needed, you can use the pre-existing one. This recipe will explore how threading in a **`Worker()`** works and how to create a custom executor.

## Getting ready

In this recipe, since we will be looking at examples, you can follow along by reading and seeing if this applies to you.

## How to do it...

Let's learn how threading works in **`WorkManager`**:

1. In order to configure **`WorkManager`** manually, you will need to specify your executor. This can be done by calling **`WorkManager.initialize()`**, then passing the context, and the configuration builder:

```
WorkManager.initialize(
    context,
    Configuration.Builder()
        .setExecutor(Executors.newFixedThreadPool(
            CONSTANT_THREAD_POOL_INT))
        .build())
```

2. In our earlier example in the previous recipe, *Understanding WorkManager state*, we spoke about a use case where we download files. These files can be in the form of PDF, JPG, PNG, or even MP4. We will look at an example that downloads content 20 times; you can specify how many times you want your content to download:

```
class GetFiles(context: Context, params: WorkerParameters) : Worker(context, params) {
    override fun doWork(): ListenableWorker.Result {
        repeat(20) {
            try {
                downloadSynchronously("Your Link")
            } catch (e: IOException) {
                return
                    ListenableWorker.Result.failure()
```

```
                }
            }
            return ListenableWorker.Result.success()
        }
    }
```

3. Currently, if we do not handle the case where the **`Worker()`** is stopped, it is good practice to ensure that it is dealt with because this is an edge case. To address this case, we need to override the **`Worker.onStopped()`** method or call **`Worker.isStopped`** where necessary to free up some resources:

```
override fun doWork(): ListenableWorker.Result {
    repeat(20) {
        if (isStopped) {
            break
        }
        try {
            downloadSynchronously("Your Link")
        } catch (e: IOException) {
            return
                ListenableWorker.Result.failure()
        }
    }
    return ListenableWorker.Result.success()
}
```

4. Finally, when you stop the worker, the result is entirely ignored until you restart the process again. We used **`CoroutineWorker`** in our earlier example since **`WorkManager`** offers support for coroutines, hence why we collected the data in a flow.

*IMPORTANT NOTE*

*Customizing your executor will require manually initializing* **`WorkManager`**.

## How it works...

There is more to learn in the **`WorkManager`** Jetpack library, and it is fair to acknowledge that it can't all be captured in just a few recipes. For instance, in some scenarios, when providing a custom threading strategy, you should use **`ListenableWorker`**.

The **`ListenableWorker`** is a class in the Android Jetpack **`WorkManager`** library that allows you to perform background work in a flexible and efficient manner. It is a subclass of the

`Worker` class and adds the ability to return a `ListenableFuture` from its `doWork()` method, which allows for easier handling of asynchronous operations.

By using `ListenableWorker`, you can create a worker that returns a `ListenableFuture` and register callbacks that will be executed when the future completes. This can be useful for tasks such as network requests or database operations that require asynchronous operations.

The `Worker`, `CoroutineWorker`, and `RxWorker` derive from this particular class. `Worker`, as mentioned, runs in the background thread; `CoroutineWorker` is highly recommended for developers using Kotlin. `RxWorker` will not be touched upon here since Rx by itself is a big topic that caters to users that develop in reactive programming.

### See also

Your application might be using Rx. In that case, there are details on how threading works in Rx and how you can use `RxWorker`. See more here: **https://developer.android.com/guide/background/persistent/threading/rxworker**.

## Understanding chaining and canceling work requests

In Android development, ensuring you properly handle your application's life cycle is crucial. Needless to say, this also applies to all background work, as a simple mistake can lead to your application draining the user's battery, memory leaks, or even causing the application to crash or suffer from an **application not responding** (**ANR**) error. This could mean terrible reviews in the Play Store, which will later affect your business and causes stress for developers. How do you ensure this issue is handled well?

This can be done by ensuring all conflicts that arise while using `WorkManager` are appropriately handled or guaranteeing the policy we touched on in the previous recipe is well coded. In this recipe, we will look into chaining and canceling work requests and how to handle long-running work properly.

Say your project requires an order by which the operation should run; **WorkManager** gives you the ability to enqueue and create a chain that specifies multiple dependent tasks, and here you can set the order in which you want the operations to occur.

## Getting ready

In this recipe, we will look at an example of how you might chain your work; since this is concept-based, we will look at the example and explain how it works.

## How to do it...

To perform chaining using **WorkManager**, follow these steps:

1. In our example, we will assume we have four unique **Worker** jobs to run in parallel. The output of these jobs will be passed to an upload **Worker**. Then, these will be uploaded to our server, like the sample project we had in the *Understanding the Jetpack WorkManager library* recipe.
2. We will have our **WorkManager()** and pass in our context; then we will call **beginWith** and pass a list of our jobs:

```
WorkManager.getInstance(context)
    .beginWith(listOf(job1, job2, job3, job4))
    .then(ourCache)
    .then(upload)
    .enqueue()
```

3. To be able to maintain or preserve all our outputs from our job, we will need to use the **ArrayCreatingInputMerger::class**:

```
val ourCache: OneTimeWorkRequest = OneTimeWorkRequestBuilder<GetDataWorker>()
    .setInputMerger(ArrayCreatingInputMerger::class)
    .setConstraints(constraints)
    .build()
```

That is about it. There is definitely more to learn, but this serves our purpose.

## How it works...

To be able to create the chain of work, we use **WorkManager.beginWith(OneTimeWorkRequest)** or use

`WorkManager.beginWith` and pass a list of the one-time work requests that you have specified.

The `WorkManager.beginWith<List<OneTimeWorkRequest>>` operations return an instance of `WorkContinuation`.

We use the `WorkContinuation.enqueue()` function to enqueue our `WorkContinuation` chain. The `ArrayCreatingInputMerger` ensures we pair each key with an array. In addition, the `ArrayCreatingInputMerger` is a class in the Android Jetpack `WorkManager` library that allows you to merge input data from multiple `ListenableWorker` instances into a single array.

Furthermore, if our keys are `unique`, we will get a result of one-element arrays. *Figure 7.3* shows the output:

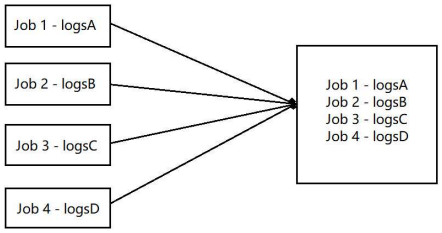


Figure 7.3 – How the array creating input merger works

If we have any colliding keys, then our values will be grouped together in our array, as in *Figure 7.4*.

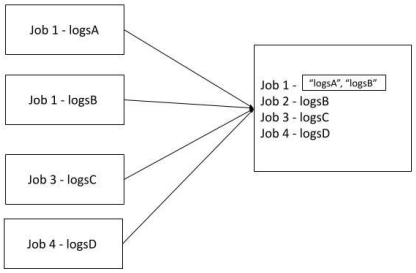


Figure 7.4 – Key collision and result

The rule of thumb is chains of work typically execute sequentially. This is reliant on the work being completed successfully. You might be wondering what happens when the job is enqueued in a chain of several work requests; just like a regular queue, all subsequent work is temporarily blocked until the first work request is completed. Think of it as *first come, first serve*.

## See also

You might be wondering how you can support long-running workers; you can learn more at **https://developer.android.com/guide/background/persistent/how-to/long-running**.

# Implementing migration from Firebase JobDispatcher to the new recommended WorkManager

In the *Understanding the Jetpack WorkManager library* recipe, we talked about other libraries that are used for scheduling and executing deferrable background work. Firebase **`JobDispatcher`** is one of the popular ones. If you have used Firebase **`JobDispatcher`**, you might know it uses the **`JobService()`** subclass as its entry point. In this recipe, we will look at how you can migrate to the newly recommended **`WorkManager`**.

## Getting ready

We will be looking at how we can migrate from **`JobService`** to **`WorkerManager`**. This might apply to your project or not. It is essential to cover it, though, due to the fact that **`WorkManager`** is highly recommended, and we all have some legacy code. However, if your project is new, you can skip this recipe.

## How to do it...

To migrate your Firebase **`JobDispatcher`** to **`WorkManager`**, follow these steps:

1. First, you will need to add the required dependency; for this, you can reference the *Understanding the Jetpack WorkManager library* recipe.
2. If you already have Firebase **`JobDispatcher`** in your project, you might have code similar to the following code snippet:

```
class YourProjectJobService : JobService() {
    override fun onStartJob(job: JobParameters):
    Boolean {
        // perform some job
```

```
            return false
        }
        override fun onStopJob(job: JobParameters):
        Boolean {
            return false
        }
    }
```

3. It is easier if your application utilizes **`JobServices()`**; then, it will map to **`ListenableWorker`**. However, if your application is utilizing **`SimpleJobService`**, then in that case, you should use **`Worker`**:

```
class YourWorker(context: Context, params: WorkerParameters) :
    ListenableWorker(context, params) {
    override fun startWork():
    ListenableFuture<ListenableWorker.Result> {
        TODO("Not yet implemented")
    }
    override fun onStopped() {
        TODO("Not yet implemented")
    }
}
```

4. If your project is using **`Job.Builder.setRecurring(true)`**, in this case, you should change it to the **`PeriodicWorkRequest`** a class offered by **`WorkManager`**. You can also specify your tag, service if the job is recurring, trigger window, and more:

```
val job = dispatcher.newJobBuilder()
    ...
    .build()
```

5. In addition, to be able to achieve what we want, we will need to input data that will act as the input data for our **`Worker`**, then build our **`WorkRequest`** with our input data and the specific constraint. You can reference the *Understanding the Jetpack WorkManager library* recipe, and finally, enqueue the Work Request.

Finally, you can create your work request as either one-time or periodic and ensure you handle any edge cases, such as canceling work.

## How it works...

In Firebase **`JobDispatcher`**, the **`JobService.onStartJob()`**, which is a function in the **`JobSccheduler`**, and **`startWork()`** are called on the main thread. In comparison, in **`WorkManager`**,

the `ListenableWorker` is the basic unit of work. In our example, `YourWorker` implements the `ListenableWorker` and returns an instance of `ListenableFuture`, which helps in signaling work completion. However, you can implement your own threading strategy based on your application's needs.

In Firebase, the `FirebaseJobBuilder` uses the `Job.Builder` serves as the Jobs metadata. In comparison, `WorkManager` uses `WorkRequest` to perform a similar role. `WorkManager` usually initializes itself by utilizing the `ContentProvider`.

# How to debug WorkManager

Any operation that requires working in the background and sometimes making network calls need proper exception handling. This is due to the fact that you do not want your users facing issues and a lack of exception handling coming back to haunt your team or you as a developer.

Hence, knowing how to debug `WorkManager` will come in handy, as this is one of those issues that might last for days if you have a bug. In this recipe, we will look at how to debug `WorkManager`.

## Getting ready

To follow this recipe, you must have completed all previous recipes of this chapter.

## How to do it...

You might encounter an issue where `WorkManager` no longer runs if it is out of sync. Follow this recipe to debug `WorkManager`:

1. To be able to set up debugging, we will need to first create a custom initialization in our `AndroidManifest.xml` file, that is, by disabling the `WorkManager` initializer:

```
<provider
    ...
    tools:node="remove"/>
```

2. After, we go ahead and set a minimum logging level to debug in our application class:

```
class App() : Application(), Configuration.Provider {
    override fun getWorkManagerConfiguration() =
        Configuration.Builder()
            .setMinimumLoggingLevel(
                android.util.Log.DEBUG)
            .build()
}
```

Once this is done, we will be able to see logs with the prefix **`WM-`** in our debug level easily, which will make our debugging work much more straightforward, and voila, we can move one step closer to solving our issue.

### How it works...

Sometimes it might be helpful just to utilize the verbose **`WorkManager`** logs to capture any anomalies. In addition, you can enable logging and use your own custom initialization. That is what we do in the first step of our recipe. Furthermore, when we declare our own custom **`WorkManager`** configuration, our **`WorkManager`** will be initialized when we call the **`WorkManager.getInstance(context)`** and not naturally at application startup.

## Testing Worker implementations

Testing your **`Worker`** implementation is crucial, as it helps ensure your code is well handled and your team follows the proper guidelines for writing great code. This will be an integration test, which means we will add our code to the **`androidTest`** folder. This recipe will look into how to add tests for your worker.

### Getting ready

To follow along with this recipe, you need to have completed all previous recipes of this chapter.

### How to do it...

Follow these steps to get started with testing **`WorkManager`**. We will look at examples in this recipe:

1. First, you need to add the testing dependency in your **`build.gradle`** file:

```
androidTestImplementation("androidx.work:work-testing:$work_version")
```

In the scenario where something in the API changes in the future, there's a stable version that you can use, and you can always find that in the documentation by following this link: **https://developer.android.com/jetpack/androidx/releases/work**.

2. We will need to set up our **`@Before`** function, as provided by Google:

```
@RunWith(AndroidJUnit4::class)
class BasicInstrumentationTest {
    @Before
    fun setup() {
        val context =
            InstrumentationRegistry.getTargetContext()
        val config = Configuration.Builder()
            .setMinimumLoggingLevel(Log.DEBUG)
            .setExecutor(SynchronousExecutor())
            .build()
        // Initialize WorkManager for instrumentation
           tests.
        WorkManagerTestInitHelper.
            initializeTestWorkManager(context, config)
    }
}
```

3. Now that we have our **`WorkManager`** set up, we can go ahead and structure our test:

```
class GetDataWorker(context: Context, parameters: WorkerParameters) : Worker(context, parameters)
    override fun doWork(): Result {
        return when(endpoint) {
            0 -> Result.failure()
            else -> Result.success(dataOutput)
        }
    }
}
```

4. You can easily test and verify the states by following this example:

```
@Test
@Throws(Exception::class)
fun testGetDataWorkerHasNoData() {
    ...
    val workInfo =
        workManager.getWorkInfoById(request.id).get()
```

```
            assertThat(workInfo.state,
                `is`(WorkInfo.State.FAILED))
        }
```

You can add more tests such as verifying when the state is successful or checking initial delays; you can also go the extra mile and test the constraints and more.

## How it works...

The library we use provides excellent support for testing `Worker`. For instance, we have `WorkManagerTestInitHelper` supplied to us through the library. Furthermore, we have the `SynchronousExecutor`, which makes our work as developers easier by ensuring synchronous writing tests is easy. Also, the issue of handling multiple threads, latches, and locks is dealt with.

In our `testGetDataWorkerHasNoData`, we create a request, then enqueue it and wait for the results. We later get the info, then assert when the state is failed, it should fail. You can also test when it is successful.

## There's more...

To test worker implementations with different variants, you can follow this link: **https://developer.android.com/guide/background/testing/persistent/worker-impl**.