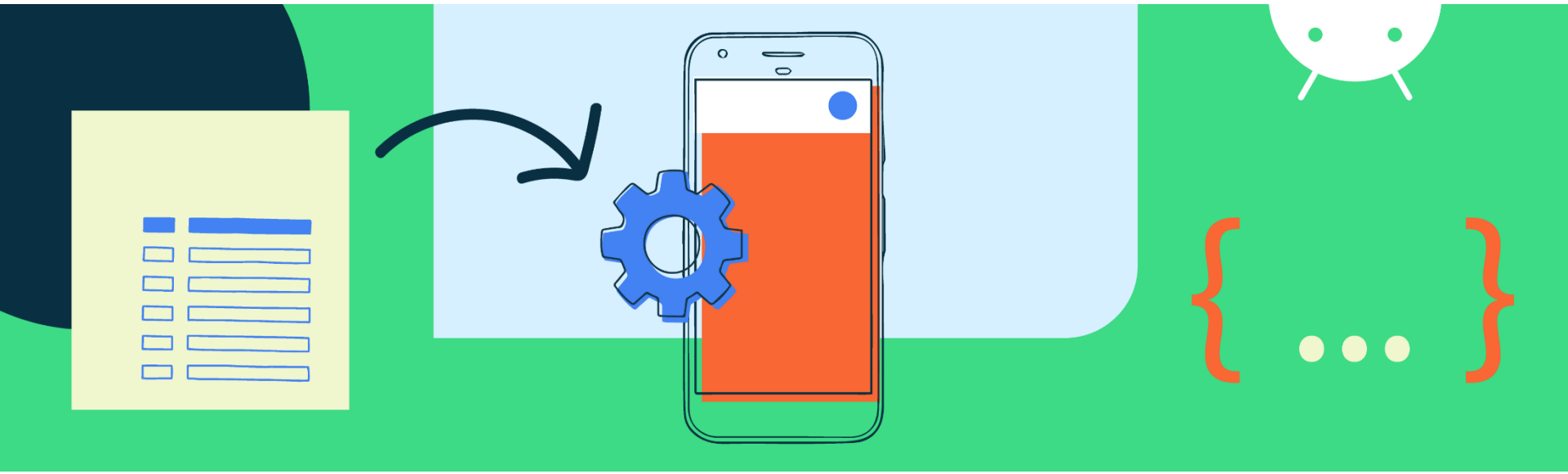
Background processing using WorkManager



WorkManager

- WorkManager is an Android library to **schedule** & **execute deferrable** background work
  - Intended for tasks that require a **guarantee** that the system will run them even if the app exits (app inactive)
- Can specify **constraints** that must be satisfied before the work is executed (e.g., only upload images to Cloud Storage when WiFi collection is available)
- Can configure **retries** if the job fails

Define work to do using Worker

- Define a unit of work to perform in the background using class that extends **`Worker`** class
  - Override **`doWork`** method

```
class DownloadWorker(context: Context, params:WorkerParameters) : CoroutineWorker(context,p
    override fun doWork(): Result {
        try {
            for (i in 0 ..3000) {
                Log.i("DownloadWorker", "Downloading $i")
            }
            val time = SimpleDateFormat("dd/M/yyyy hh:mm:ss aa")
            val currentDate = time.format(Date())
            Log.i("DownloadWorker","Completed $currentDate")
            return Result.success()
        } catch (e:Exception){
            return Result.failure()
        }
    }
}
```

Launch a worker

- Create a **`OneTimeWorkRequest`**
- Then **enqueue** the request

```
val downloadRequest= OneTimeWorkRequestBuilder<DownloadWorker>().build()

WorkManager.getInstance(applicationContext)
           .enqueue(downloadRequest)
```

Schedule

- Use **`PeriodicWorkRequest`** to schedule a work to repeat periodically

```
val periodicWorkRequest = PeriodicWorkRequestBuilder<DownloadWorker>
                                    (15, TimeUnit.MINUTES).build()

WorkManager.getInstance(applicationContext).enqueue(periodicWorkRequest)
```

Define Constraints

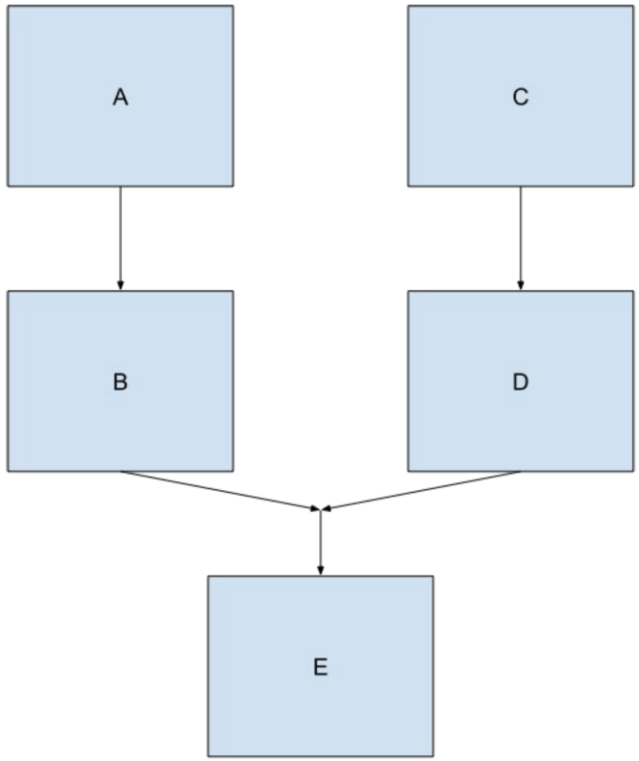
- You can define constraints that must be met before the job starts:
  - Network connectivity
  - Device charging

```
val constraints = Constraints.Builder()
                              .setRequiresCharging(true)
                              .setRequiredNetworkType(NetworkType.CONNECTED)
                              .build()

val uploadRequest = OneTimeWorkRequestBuilder<UploadWorker>()
                                        .setConstraints(constraints)
                                        .setInputData(data)
                                        .build()
```

Work Chaining

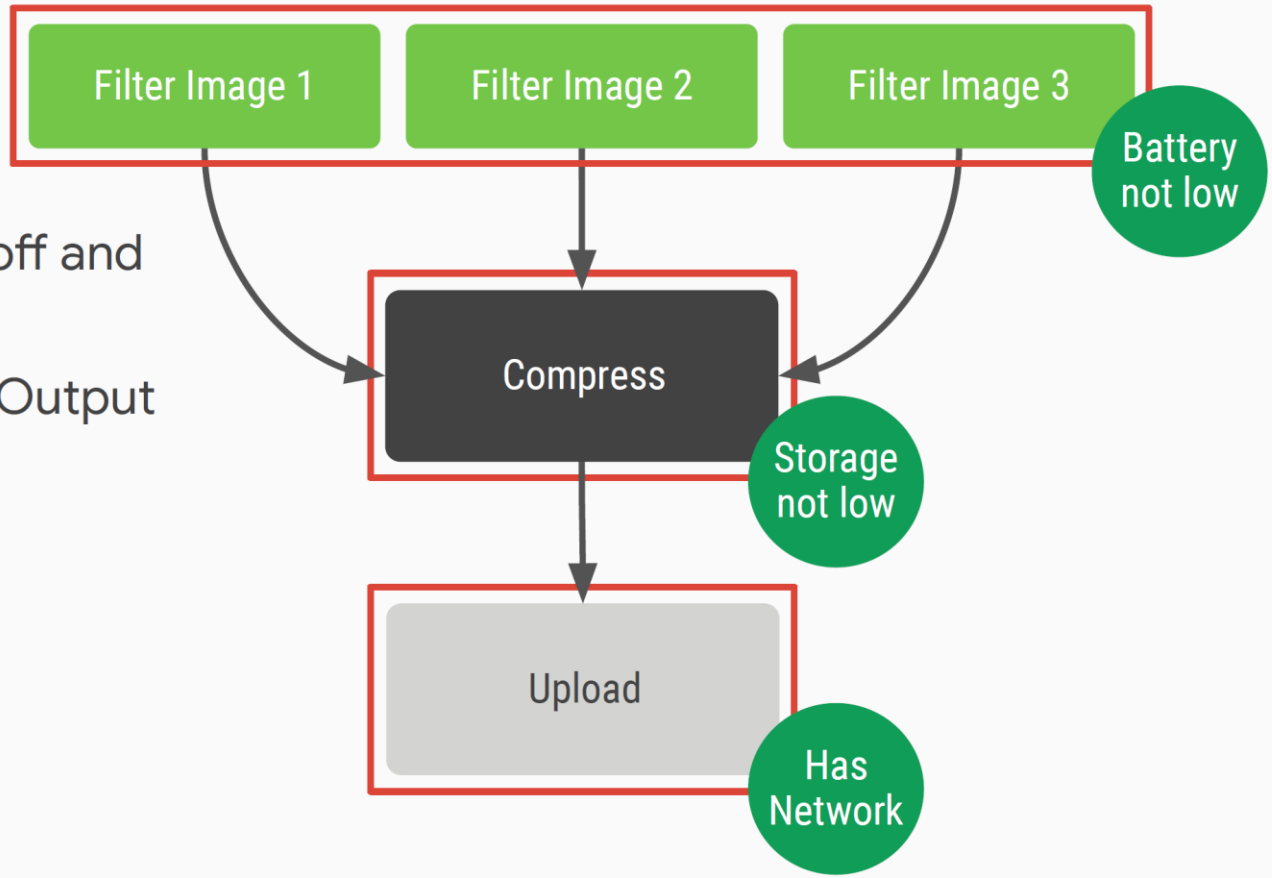
- Orchestration of multiple jobs. E.g.,
  - B runs after A
  - D runs after C
  - E runs after B and D are completed




```
val parallelWorks = listOf(downloadRequest, filterRequest)
workManager.beginWith(parallelWorks)
           .then(compressRequest)
           .then(uploadRequest)
           .enqueue()
```

Example usage

- Asynchronous one-off and periodic tasks
- Chaining with Input/Output
- Constraints

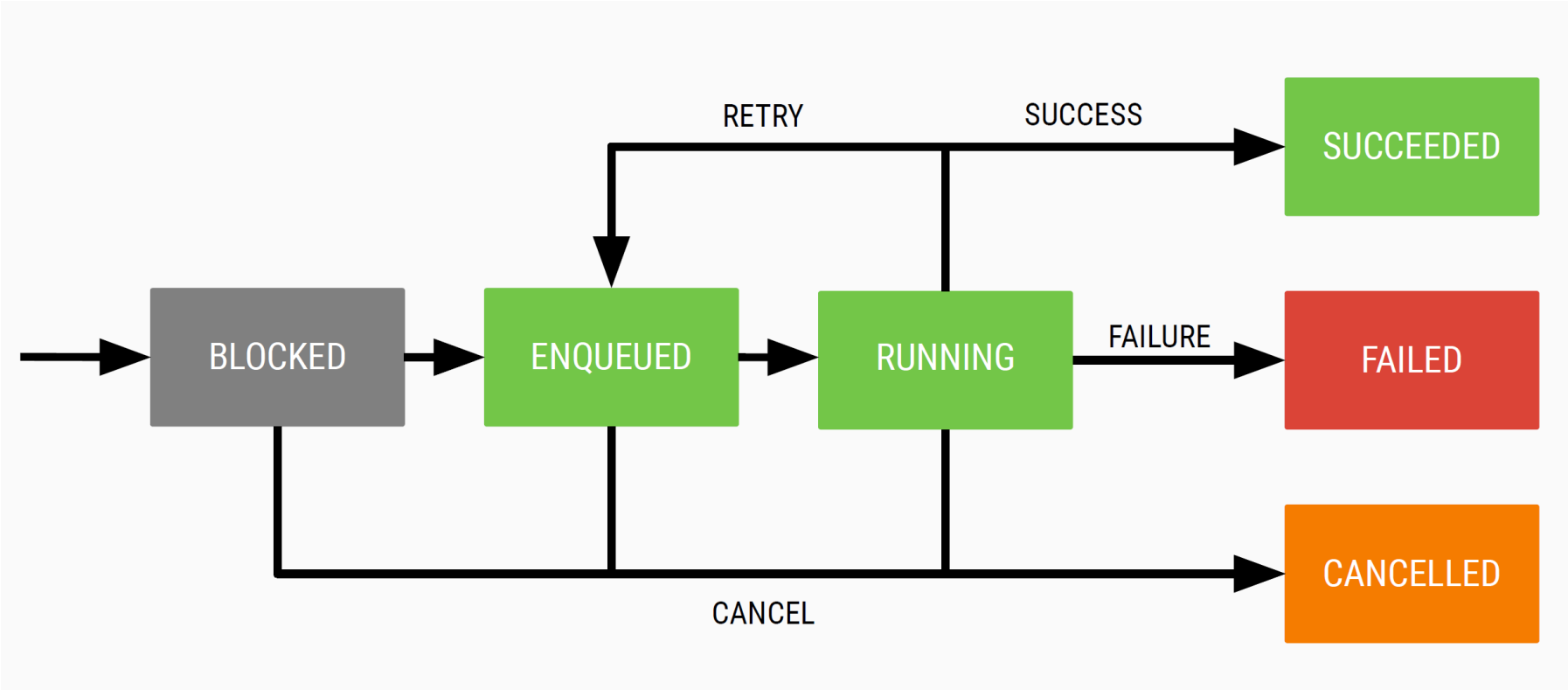


Monitor

- Monitor status → LiveData providing job status
- Use **`.getWorkInfoByIdLiveData`** to observe the work progress

```
workManager.getWorkInfoByIdLiveData(uploadRequest.id)
        .observe(this, Observer {
            textView.text = it.state.name
            if(it.state.isFinished){
                val data = it.outputData
                val message = data.getString(AppKeys.CURRENT_DATE)
                Toast.makeText(applicationContext, message, Toast.LENGTH_LONG).show()
            }
        })
```

Life of OneTime Work



Cancel Work

```
val save = OneTimeWorkRequestBuilder<SaveImageWorker>()
        .addTag(TAG_SAVE)
        .build()


WorkManager.getInstance().cancelWorkById(save.id)
```

```
val save = OneTimeWorkRequestBuilder<SaveImageWorker>()
        .addTag(TAG_SAVE)
        .build()


WorkManager.getInstance().cancelWorkById(save.id)
WorkManager.getInstance().cancelAllWorkByTag(TAG_SAVE)
```

Configure retries

- If you require that WorkManager retry failed work, you can return **Result.retry()** from your worker. Your work is then rescheduled according to a **backoff delay** and **backoff policy**.

```
val uploadRequest = OneTimeWorkRequestBuilder<UploadWorker>()
                    .setBackoffCriteria(
                        BackoffPolicy.LINEAR,
                        OneTimeWorkRequest.MIN_BACKOFF_MILLIS,
                        TimeUnit.MILLISECONDS)
                    .build()
```

Coroutines + WorkManager

- Use **`CoroutineWorker`** to use coroutines in a Worker

```
class MyWork(context: Context, params: WorkerParameters) :
        CoroutineWorker(context, params) {
override suspend fun doWork(): Result = withContext(Dispatchers.IO) {

return try {
            // Do something async

            Result.success()
        } catch (error: Throwable) {
            Result.failure()
        }
    }

}
```

Resources

- Getting started with WorkManager
  - https://developer.android.com/topic/libraries/architecture/workmanager/basics
  - https://developer.android.com/topic/libraries/architecture/workmanager
- WorkManager codelab
  - https://developer.android.com/codelabs/android-workmanager