Unit 7 — WorkManager (Background Work)

Mobile Application Development — Lecture 7

Masoud Hamad

State University of Zanzibar (SUZA)

Semester II, 2025/2026

Outline

Background Work on Android

Apps often need to do work that:

- Does not require the user to be watching
- Must run even if the app is closed
- Should survive device reboots
- Requires specific conditions (network, charging, storage)

Examples:

- Sync user data with a server
- Download fresh content nightly
- Upload logs when Wi-Fi becomes available
- Back up photos when charging

Why WorkManager?

API	Best for
Coroutines (`viewModelScope`)	short async work while app alive
`AlarmManager`	exact-time wake-ups
`JobScheduler`	deferrable work (API 21+)
WorkManager	**guaranteed deferrable work**

WorkManager is Google's **recommended API** for persistent background work that needs to run reliably, even if the app is killed or the device reboots.

Dependency

```
// build.gradle.kts
implementation("androidx.work:work-runtime-ktx:2.9.0")
```

WorkManager Concepts

- **Worker** — defines the work to do (implements `doWork()`)
- **WorkRequest** — instructions: which Worker, constraints, inputs
  - `OneTimeWorkRequest` — runs once
  - `PeriodicWorkRequest` — runs repeatedly (min 15 min)
- **WorkManager** — schedules and runs WorkRequests
- **Constraints** — network, charging, storage, battery-not-low
- **Data** — typed key/value input and output

CoroutineWorker

```
class UploadWorker(
    context: Context,
    params: WorkerParameters
) : CoroutineWorker(context, params) {

    override suspend fun doWork(): Result {
        val fileUri = inputData.getString("file_uri") ?: return Result.failure()
        return try {
            uploadFile(fileUri)                          // your suspend fn
            val output = workDataOf("url" to "https://...")
            Result.success(output)
        } catch (e: IOException) {
            Result.retry()                               // will retry with backoff
        }
    }
}
```

Result Types

- `Result.success()` — work completed; may include output `Data`
- `Result.failure()` — work failed permanently; do not retry
- `Result.retry()` — retry later with exponential backoff

Chained work

Output of one worker can be passed as input to the next — useful for multi-step pipelines.

One-Time Work

```
val input = workDataOf("file_uri" to uri.toString())

val request = OneTimeWorkRequestBuilder<UploadWorker>()
    .setInputData(input)
    .setConstraints(
        Constraints.Builder()
            .setRequiredNetworkType(NetworkType.CONNECTED)
            .setRequiresCharging(false)
            .build()
    )
    .build()

WorkManager.getInstance(context).enqueue(request)
```

Periodic Work

```
val daily = PeriodicWorkRequestBuilder<SyncWorker>(
    repeatInterval = 1, repeatIntervalTimeUnit = TimeUnit.DAYS
)
.setConstraints(
    Constraints.Builder()
        .setRequiredNetworkType(NetworkType.UNMETERED)    // Wi-Fi only
        .setRequiresBatteryNotLow(true)
        .build()
)
.build()

WorkManager.getInstance(context).enqueueUniquePeriodicWork(
    "daily_sync",
    ExistingPeriodicWorkPolicy.KEEP,
    daily
)
```

Minimum interval: **15 minutes**.

Chaining WorkRequests

```
WorkManager.getInstance(context)
    .beginWith(compressRequest)
    .then(uploadRequest)
    .then(cleanupRequest)
    .enqueue()
```

Output of each worker is passed to the next. Use `beginUniqueWork` if you want only one chain with a given name at a time.

Observing Progress in Compose

```
val workInfo by WorkManager.getInstance(context)
    .getWorkInfoByIdLiveData(request.id)
    .observeAsState()

when (workInfo?.state) {
    WorkInfo.State.ENQUEUED  -> Text("Waiting...")
    WorkInfo.State.RUNNING   -> CircularProgressIndicator()
    WorkInfo.State.SUCCEEDED -> {
        val url = workInfo?.outputData?.getString("url")
        Text("Uploaded: $url")
    }
    WorkInfo.State.FAILED    -> Text("Failed")
    else -> {}
}
```

Constraints Reference

Constraint	Description
`setRequiredNetworkType`	`CONNECTED`, `UNMETERED`, `METERED`
`setRequiresCharging`	device must be charging
`setRequiresBatteryNotLow`	battery $> 15\%$
`setRequiresStorageNotLow`	sufficient free space
`setRequiresDeviceIdle`	device must be idle

A worker with unmet constraints stays `ENQUEUED` until conditions are met.

Backoff & Retry

- `Result.retry()` triggers exponential backoff by default
- Customise with `setBackoffCriteria(BackoffPolicy.EXPONENTIAL, 30, TimeUnit.SECONDS)`
- Max attempts controllable; WorkManager persists state across reboots

Expedited Work

For time-sensitive tasks that should run as soon as possible:

```
val request = OneTimeWorkRequestBuilder<QuickWorker>()
    .setExpedited(OutOfQuotaPolicy.RUN_AS_NON_EXPEDITED_WORK_REQUEST)
    .build()
```

- Runs immediately, ignoring most constraints
- Must finish in ∼10 min
- Shows a foreground-service notification on older Android versions

Summary

- WorkManager = Google's recommended API for **reliable deferrable background work**
- `Worker` / `CoroutineWorker` defines the task
- `WorkRequest` = Worker + constraints + inputs
- Survives app kills and device reboots
- Observable state (`ENQUEUED`, `RUNNING`, `SUCCEEDED`, `FAILED`)

Next: Unit 8 — interop between Views (XML) and Compose.

Questions?