**Section 33 – WorkManager – Android Jetpack**

**1. Key Concepts**

1. **What is WorkManager?**
   * A **Jetpack** library for running **reliable background tasks**.
   * Designed for work that **must run even if the app is closed or the device restarts**.
   * Handles:
     + Device restarts (persistence)
     + System optimizations (Doze mode, battery saver)
     + Automatic retries on failure
2. **When to Use**
   * Uploading logs/data to server
   * Syncing data periodically
   * Applying filters to images/videos in background
   * Sending analytics
   * Scheduled notifications
3. **When Not to Use**
   * Immediate foreground work → Use coroutines / Thread / Executor
   * Continuous tasks like music streaming → Use ForegroundService
4. **Execution Model**
   * **Worker** → Defines the actual task in doWork() method.
   * **WorkRequest** → Tells WorkManager *what* to run and *when*.
   * **WorkManager** → Schedules and executes.
5. **Types of Work**
   * **OneTimeWorkRequest** → Runs once.
   * **PeriodicWorkRequest** → Runs at fixed intervals (min. 15 min).
6. **Constraints**
   * Rules for when work should run (charging, network type, battery level).
7. **Data Passing**
   * **InputData** → Send values to Worker.
   * **OutputData** → Return results to caller.
8. **Work States**
   * ENQUEUED, RUNNING, SUCCEEDED, FAILED, CANCELLED

**2. Step-by-Step Implementation**

**Step 1 – Add Dependency**

dependencies {

implementation "androidx.work:work-runtime:2.9.0" // Java

// For Kotlin coroutines:

implementation "androidx.work:work-runtime-ktx:2.9.0"

}

**Step 2 – Create a Worker Class**

**Java:**

public class MyWorker extends Worker {

// Constructor: receives app context and Worker parameters

public MyWorker(@NonNull Context context, @NonNull WorkerParameters workerParams) {

super(context, workerParams);

}

@NonNull

@Override

public Result doWork() {

// Get input data (if provided)

int limit = getInputData().getInt("count\_limit", 0);

// Background task logic

for (int i = 1; i <= limit; i++) {

Log.d("WorkManagerDemo", "Counting: " + i);

}

// Prepare output data to send back

Data outputData = new Data.Builder()

.putString("result\_msg", "Counting completed up to " + limit)

.build();

// Indicate success + send output data

return Result.success(outputData);

}

}

**Kotlin equivalent:**

class MyWorker(context: Context, params: WorkerParameters) : Worker(context, params) {

override fun doWork(): Result {

val limit = inputData.getInt("count\_limit", 0)

for (i in 1..limit) {

Log.d("WorkManagerDemo", "Counting: $i")

}

val outputData = workDataOf("result\_msg" to "Counting completed up to $limit")

return Result.success(outputData)

}

}

**Step 3 – Create a WorkRequest**

**Java:**

Data input = new Data.Builder()

.putInt("count\_limit", 500)

.build();

Constraints constraints = new Constraints.Builder()

.setRequiresCharging(true) // Only when charging

.setRequiredNetworkType(NetworkType.CONNECTED) // Needs internet

.build();

OneTimeWorkRequest workRequest =

new OneTimeWorkRequest.Builder(MyWorker.class)

.setInputData(input)

.setConstraints(constraints)

.build();

**Step 4 – Enqueue the Work**

WorkManager.getInstance(this).enqueue(workRequest);

**Step 5 – Observe Work Status**

WorkManager.getInstance(this)

.getWorkInfoByIdLiveData(workRequest.getId())

.observe(this, workInfo -> {

if (workInfo != null) {

Log.d("WorkManagerDemo", "Status: " + workInfo.getState());

if (workInfo.getState().isFinished()) {

String result = workInfo.getOutputData().getString("result\_msg");

Toast.makeText(this, result, Toast.LENGTH\_SHORT).show();

}

} });

**Step 6 – Periodic Work**

PeriodicWorkRequest periodicRequest =

new PeriodicWorkRequest.Builder(MyWorker.class, 15, TimeUnit.MINUTES)

.build();

WorkManager.getInstance(this)

.enqueue(periodicRequest);

⛔ **Minimum repeat interval** is 15 minutes — system restriction.

**Step 7 – Cancel Work**

WorkManager.getInstance(this)

.cancelWorkById(workRequest.getId());

**3. Tools, Libraries, APIs**

* **WorkManager** – Android Jetpack component for background work.
* **Constraints API** – Battery, network, charging state conditions.
* **Data API** – Send/receive key-value data.
* **LiveData** – Observe work progress.
* **Logcat** – Debugging.

**4. Best Practices**

* Always make Workers **idempotent** (safe to re-run without breaking state).
* Use **constraints** to avoid wasting battery/data.
* Chain dependent tasks using:
* WorkManager.getInstance(this)
* .beginWith(workA)
* .then(workB)
* .enqueue();
* Use **unique work names** for periodic tasks to prevent duplicates:
* enqueueUniquePeriodicWork("SyncJob", ExistingPeriodicWorkPolicy.KEEP, periodicRequest);
* For tasks longer than ~10 minutes, use **ForegroundWorker** with notifications.
* For network calls, integrate Retrofit inside doWork().
* For persistence, combine WorkManager with Room DB.

**5. Part B – Extra Knowledge Not in the Section**

1. **Retry with Backoff Policy**
2. .setBackoffCriteria(
3. BackoffPolicy.LINEAR,
4. 10, TimeUnit.MINUTES
5. )
6. **Work Chaining (Parallel + Sequential)**
   * Parallel: WorkManager.getInstance().beginWith(Arrays.asList(work1, work2))
   * Sequential: .then(work3)
7. **Testing WorkManager**
   * Use WorkManagerTestInitHelper in local unit tests.
8. **Foreground Service Hybrid**
   * For long tasks with guaranteed execution + visible progress.
9. **Doze Mode Awareness**
   * WorkManager automatically adapts, but heavy tasks might be delayed.
10. **Replacing AlarmManager**
    * For most background jobs, WorkManager is the modern replacement.
11. **Kotlin + CoroutineWorker**
12. class MyWorker(appContext: Context, params: WorkerParameters) : CoroutineWorker(appContext, params) {
13. override suspend fun doWork(): Result {
14. // Suspend-friendly API calls
15. return Result.success()
16. }
17. }
18. **Progress Updates**
    * Use setProgressAsync() in Worker and observe progress LiveData in UI.