WorkManager

Clever Delegate for Deferrable background tasks

Divya Jain
Mobile Developer
@divyajain2405

APRIL 08+09
DROIDCON BOS19

Task

Task

Task

Task

Task

Task

Task

Task

Task

Task

Challenge?

- Main thread responsible for multiple tasks
- Too much work == undesired user experience
- More than few milliseconds ? -> Need a background thread
- Various Criterias for these tasks
- Mobile device has limited resources

Background Restrictions

- Doze Mode & App Standby (6.0)
- Doze on the Go (7.0)
- Limited background behavior (8.0)
- App Standby Buckets (9.0)

Developer Solutions!

Download Manager

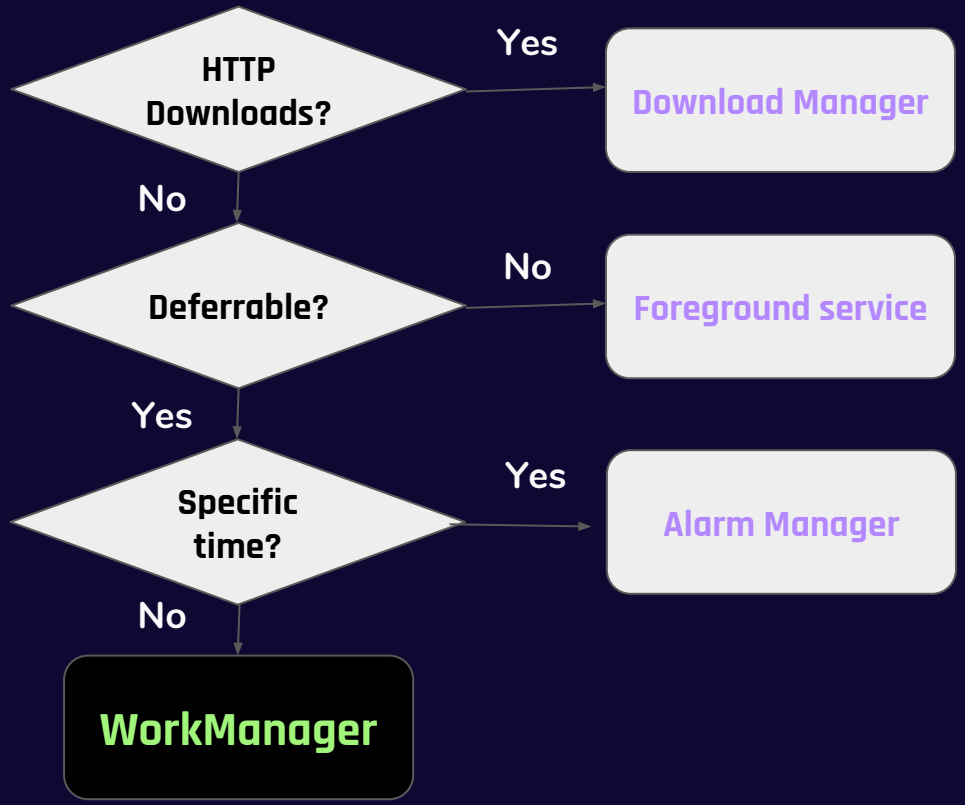
Alarm Manager

Schedule Jobs/ Workmanager

Multiple Threads

Services

Ask Questions - Choose the right solution



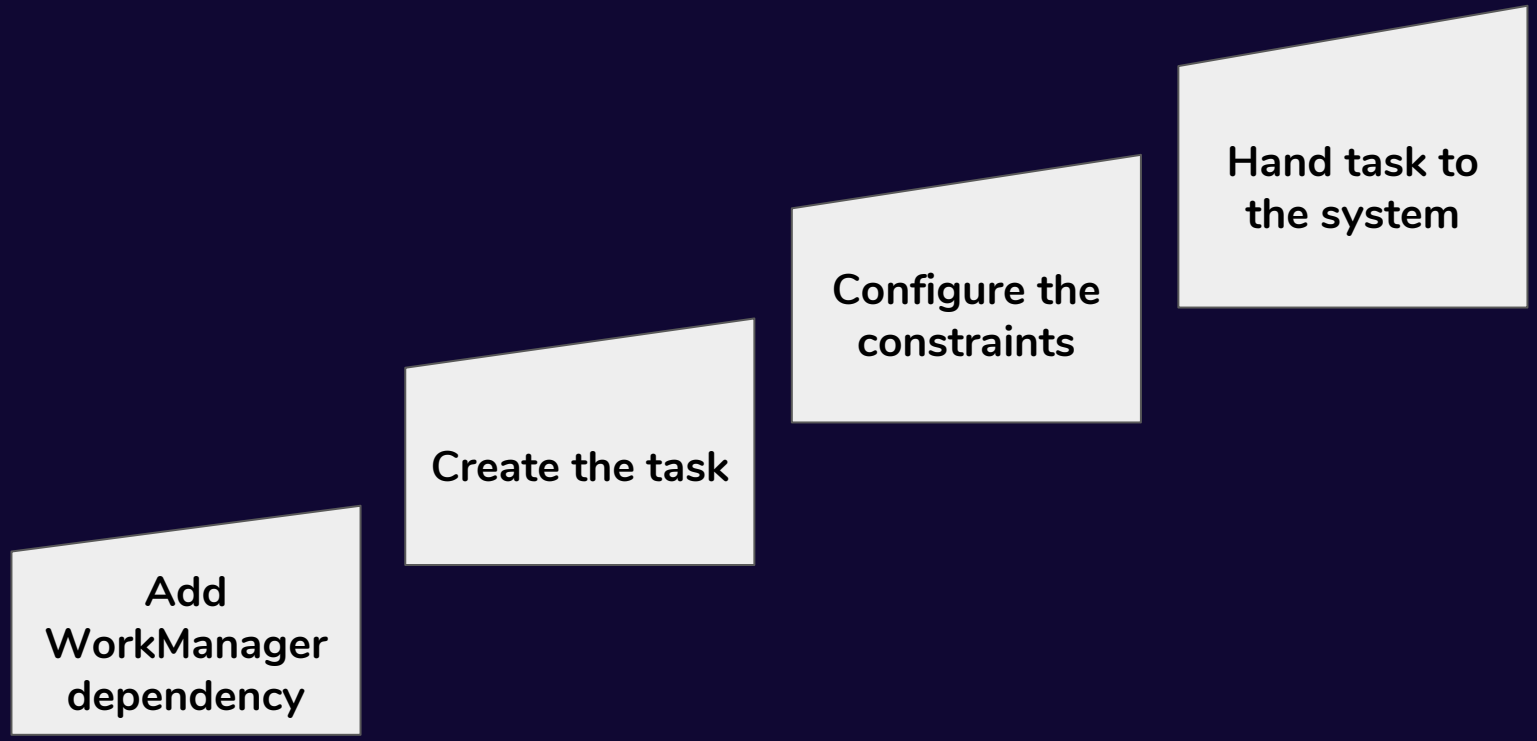
Basis for Choosing WorkManager

- Deferrable
- Require Specific system conditions
- No Particular time
- Reliable execution

WorkManager features:

- Backward **compatibility**
- Specify work **constraints**
- Schedule **one time / recurring** jobs
- **Manage** & **monitor** the scheduled work
- **Certainty** that task will execute
- **Optimized** use of System **resources**

How do I start?



Adding workmanager to Android project

```
dependencies {
   def work_version = 2.0.0

   // (Java only)
   implementation "androidx.work:work-runtime:$work_version"

   // Kotlin + coroutines
   implementation "androidx.work:work-runtime-ktx:$work_version"

   // optional - RxJava2 support
   implementation "androidx.work:work-rxjava2:$work_version"
   // optional - Test helpers
   androidTestImplementation "androidx.work:work-testing:$work_version"
 }
```

** WorkManager requires compileSdk version 28+*

Create the background task

Extend **Worker** ⟶ Override **doWork()** ⟶ Get **Result**

```
class UploadWorker(appContext: Context, workerParams: WorkerParameters)
    : Worker(appContext, workerParams) {

    override fun doWork(): Result {
        // Do the work here--in this case, sync to backend.

        syncToServer()

        // Indicate whether the task finished successfully with the Result
        return Result.success()
    }
}
```

Configure the task

WorkRequest

- OneTimeWorkRequest - for one time tasks
- PeriodicTimeWorkRequest - for recurring tasks

```
val syncWorkRequest = OneTimeWorkRequestBuilder<UploadWorker>()
        .build()

val syncImagesRequest =
     PeriodicWorkRequestBuilder<UploadWorker>(1, TimeUnit.HOURS)
        .build()
```

Add Constraints

Constraints.Builder()

```
// Create a Constraints object that defines when the task should run
val constraints = Constraints.Builder()
        .setRequiredNetworkType(NetworkType.METERED)
        .setRequiresCharging(true)
        .build()

// ...then create a OneTimeWorkRequest that uses those constraints
val syncWorkRequest = OneTimeWorkRequestBuilder<UploadWorker>()
        .setConstraints(constraints)
        .build()
```

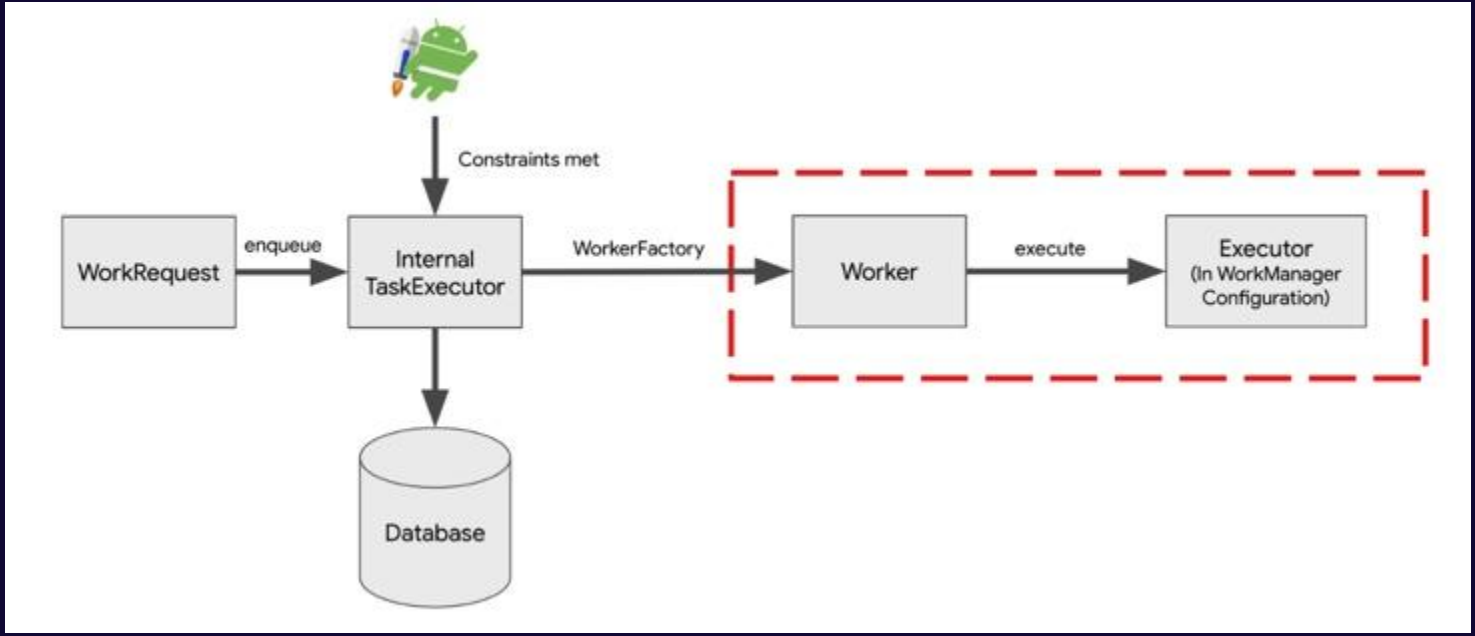
Hand the task off to the system

Schedule the WorkRequest with the WorkManager using

enqueue()

```
WorkManager.getInstance().enqueue(uploadWorkRequest)
```

Internal mechanism



Input/Output for the task

Data - **Key value** pair of primitive data types and Arrays

```
// workDataOf (part of KTX) converts a list of pairs to a [Data] object.
val imageData = workDataOf(Constants.KEY_IMAGE_URI to imageUriString)

val uploadWorkRequest = OneTimeWorkRequestBuilder<UploadWorker>()
        .setInputData(imageData)
        .build()
```

```kotlin
class UploadWorker(appContext: Context, workerParams: WorkerParameters)
    : Worker(appContext, workerParams) {

    override fun doWork(): Result {

            // Get the input
            val imageUriInput = getInputData().getString(Constants.KEY_IMAGE_URI)

            // Do the work
            val response = uploadFile(imageUriInput)

            // Create the output of the work
            val outputData = workDataOf(Constants.KEY_IMAGE_URL to response.imageUrl)

            // Return the output
            return Result.success(outputData)

    }
}
```

Delays and Retries

Initial Delay : Minimum delay before task execution

```
val syncWorkRequest = OneTimeWorkRequestBuilder<SyncWorker>()
        .setInitialDelay(15, TimeUnit.MINUTES)
        .build()
```

Retries and BackOff Policy : Result.retry()

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

Tagging of Tasks

Logically group a set of tasks

```
WorkRequest.Builder.addTag(string)

val cacheCleanupTask =
        OneTimeWorkRequestBuilder<CacheCleanupWorker>()
    .setConstraints(constraints)
    .addTag("cleanup")
    .build()
```

Work States

- **BLOCKED** - Prerequisite work not completed
- **ENQUEUED** - Eligible to execute when Constraints & timing are met
- **RUNNING** - In the process of execution
- **SUCCEEDED** - Worker has returned **Result.success()**
- **FAILED** - Worker has returned **Result.failed()**
- **CANCELLED** - User explicitly cancelled the task

How to observe work status

WorkInfo - In the form of LiveData

- Id
- Tags
- Current state
- Output data

```
WorkManager.getInstance().getWorkInfoByIdLiveData(uploadWorkRequest.id)
        .observe(lifecycleOwner, Observer { workInfo ->
            if (workInfo != null && workInfo.state ==
WorkInfo.State.SUCCEEDED) {
                displayMessage("Work finished!")
            }
        })
```

Ways to retrieve WorkInfo

- WorkRequest Id

```
WorkManager.getWorkInfoById(UUID)
WorkManager.getWorkInfoByIdLiveData(UUID)
```

- Tag

```
WorkManager.getWorkInfosByTag(String)
WorkManager.getWorkInfosByTagLiveData(String)
```

- Unique Name

```
WorkManager.getWorkInfosForUniqueWork(String)
WorkManager.getWorkInfosForUniqueWorkLiveData(String)
```

Chaining Work

Create/enqueue chain of multi dependent tasks & the order of execution

```
WorkManager.getInstance()
    // Candidates to run in parallel, returns instance of
WorkContinuation
    .beginWith(listOf(parallel1, parallel2, parallel3))
    // Dependent work (only runs after all previous work in
chain)
    .then(task2)
    .then(task3)
    // enqueue to hand off the task to the system()
    .enqueue()
```

Output to Input in Chained Work

Output of task -> **Input** of the next dependent task

InputMerger (s)

- **OverwritingInputMerger** : overwrites the keys in case of conflicts
- **ArrayCreatingInputMerger** : merges the inputs, creates Arrays

```
val syncWorkRequest = OneTimeWorkRequestBuilder<SyncWorker>()
        .setInputMerger(ArrayCreatingInputMerger::class)
        .setConstraints(constraints)
        .build()
```

Unique work

Only **one chain of work** with a unique, human readable & developer specified

Name

```
WorkManager.enqueueUniqueWork(String, ExistingWorkPolicy, OneTimeWorkRequest)
WorkManager.enqueueUniquePeriodicWork(String, ExistingPeriodicWorkPolicy,
PeriodicWorkRequest)
```

ExistingWorkPolicy == Resolution Policy

REPLACE/KEEP/APPEND

Cancelling work

```
WorkManager.cancelWorkById(workRequest.id)

WorkManager.cancelAllWorkByTag(String)

WorkManager.cancelUniqueWork(String)
```

Solving a ticketing problem with WorkManager

- User is able to buy tickets to an event and proceed to the checkout page
- User is able to apply “Credits” to the order which updates the checkout cost
- User leaves the checkout page before confirming the purchase, order needs to be reset to release the credits
- Each of the above steps make a network API call

```
checkoutAPI.savePurchase()
checkoutAPI.applyCredits(credits)
checkoutAPI.resetPurchase(purchaseId)
```

User leaves the checkout page before confirming when no network connection?

```
class ResetPurchaseWorker(context: Context, workerParams: WorkerParameters) : Worker(context,
workerParams) {

    override fun doWork(): Result {
        val response = checkoutApi
                .resetPurchase(purchaseId)
                .execute()

        if (response.isSuccessful) {
            return Result.SUCCESS
        } else {
            if (response.code() in (500..599)) {
                // try again if there is a server error
                return Result.RETRY
            }
            return Result.FAILURE
        }
    }
}
```

```
fun onUserExit() {
    val constraints =
Constraints.Builder().setRequiredNetworkType(NetworkType.CONNECTED).build()

    val request: OneTimeWorkRequest =
            OneTimeWorkRequestBuilder<ResetPurchaseWorker>()
                    .setConstraints(constraints)
                    .addTag("reset-purchase")
                    .setBackoffCriteria(BackoffPolicy.EXPONENTIAL, 30, TimeUnit.SECONDS)
                    .build()

    WorkManager.getInstance()
            .beginUniqueWork(ResetPurchaseWorker.tag, ExistingWorkPolicy.KEEP, request)
            .enqueue()
}
```

```
override fun onResume(){
    with(WorkManager.getInstance()) {
        cancelAllWorkByTag("reset-purchase")
        getStatusesByTag("reset-purchase").observe(this@CheckoutFragment, Observer {
statusList ->
            if (statusList == null || statusList.isEmpty()) {
                savePurchase()
                return@Observer
            }

            val allWorkersFinished = statusList.all { status -> status.state.isFinished }

            if (allWorkersFinished) {
                savePurchase()
            }
        })
    }
}
```

WorkManager is cool!

- Backward compatibility with different OS Versions
- Supports one time/ recurring tasks
- Supports complex chain of tasks with input / outputs handled
- Provides the ability to Set constraints on task execution
- Follows best health practices for the system, optimizations
- Guarantees Execution, even if app or device restarts


Thanks!