Andhra Pradesh State Skill Development Corporation

Skill AP
APSSDC

ANDROID APPLICATION DEVELOPMENT

JOB SCHEDULER

JobScheduler

Introduction

JobScheduler was introduced in Lollipop, and is pretty cool because it performs work based on conditions, not on time.

JobScheduler is guaranteed to get your job done, but since it operates at the system level, it can also use several factors to intelligently schedule your background work to run with the jobs from other apps. This means we can minimize things like radio use, which is a clear battery win. And as of API 24, JobScheduler even considers memory pressure, which is a clear overall win for devices and their users.

Our goal with JobScheduler was to find a way for the system to shoulder part of the burden of creating performant apps. As a developer, you do your part to create an app that doesn't freeze, but that doesn't always translate to the battery life of the device being healthy. So, by introducing JobScheduler at the system level, we can focus on batching similar work requests together, which results in a noticeable improvement for both battery and memory.

How to use JobScheduler

When talking about "using JobScheduler," the conversation is really addressing three separate pieces at once.

First, consider your job.

The work you are wanting to schedule should be defined in a JobService. Your JobService is actually going to be a Service that extends the JobService class. This is what enables the system to perform your work for you, regardless of whether your app is active. The most convenient part of this structure is that you can write multiple JobServices, where each one defines a different task that should be performed at some point for your app. This helps you modularize your code base, and makes code maintenance much simpler. So that's a win for app architecture.

Since you are extending another class with your JobService, you will need to implement a few required methods:

- **onStartJob()** is called by the system when it is time for your job to execute. If your task is short and simple, feel free to implement the logic directly in onStartJob() and return false when you are finished, to let the system know that all work has been completed. But if you need to do a more complicated task, like connecting to the network, you'll want to kick off a background thread and return true, letting the system know that you have a thread still running and it should hold on to your wakelock for a while longer.
  Note: Your JobService will run on the main thread. That means that you need to manage any asynchronous tasks yourself (like using a Thread or AsyncTask to open a network connection, and then returning true) within your onStartJob() method.
  Often, your onStartJob() will be small, as it simply kicks off something else. For example, it is essentially just initializing an AsyncTask.

```
@Override
public boolean onStartJob(final JobParameters params) {
 mDownloadArtworkTask = new DownloadArtworkTask(this) {
  @Override
  protected void onPostExecute(Boolean success) {
   jobFinished(params, !success);
  }
 };
 mDownloadArtworkTask.execute();
 return true;
}
```

- **jobFinished()** is not a method you override, and the system won't call it. That's because you need to be the one to call this method once your service or thread has finished working on the job. If your onStartJob() method kicked off another thread and then returned true, you'll need to call this method from that thread when the work is complete. This is how to system knows that it can safely release your wakelock. If you forget to call jobFinished(), your app is going to look pretty guilty in the battery stats lineup.
  jobFinished() requires two parameters: the current job, so that it knows which wakelock can be released, and a boolean indicating whether you'd like to reschedule the job. If you pass in true, this will kick off the JobScheduler's exponential backoff logic for you.

```
@Override
public boolean onStartJob(final JobParameters params) {
 mDownloadArtworkTask = new DownloadArtworkTask(this) {
  @Override
  protected void onPostExecute(Boolean success) {
   jobFinished(params, !success);
  }
 };
 mDownloadArtworkTask.execute();
 return true;
}
```

- **onStopJob()** is called by the system if the job is cancelled before being finished. This generally happens when your job conditions are no longer being met, such as when the device has been unplugged or if WiFi is no longer available.
  So use this method for any safety checks and clean up you may need to do in response to a half-finished job. Then, return true if you'd like the system to reschedule the job, or false if it doesn't matter and the system will drop this job. (For the most part, you shouldn't encounter this situation too often, but if you're having a lot of trouble, consider trying to shorten your job.For example, if your download never finishes, have your server split the download into smaller packets that can be retrieved quicker.)

```
@Override
public boolean onStopJob(final JobParameters params) {
 if (mDownloadArtworkTask != null) {
  mDownloadArtworkTask.cancel(true);
 }
```

```
    return true;
  }
```

As with any service, you'll need to add this one to your AndroidManifest.xml

What's different, though, is that you need to add a permission that will allow JobScheduler to call your jobs, and be the ~only one~ that can access your JobService.

```
<service
  android:name=".sync.DownloadArtworkJobService"
  android:permission="android.permission.BIND_JOB_SERVICE"
  android:exported="true"/>
```

Second, consider the conditions that must be true to run your job.

Remember: the great advantage to JobScheduler is that it doesn't perform work solely based on time, but rather based on conditions. (Which means you no longer need a repeating alarm to go off every few hours so that you can check to see if now is a good time to sync with your server.) You can define these conditions through the JobInfo object. To build that JobInfo object, you need two things every time (and then the criteria are the bonus bits): a job number — to help you distinguish which job this is — and your JobService.

```
JobScheduler jobScheduler =
   (JobScheduler) getSystemService(Context.JOB_SCHEDULER_SERVICE);
jobScheduler.schedule(new JobInfo.Builder(LOAD_ARTWORK_JOB_ID,
   new ComponentName(this, DownloadArtworkJobService.class))
   .setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY)
   .build());
```

Let's take a moment and talk about the potential criteria you can include in your JobInfo object.

- **Network type (metered/unmetered)** If your job requires network access, you must include this condition. You can specify a metered or unmetered network, or any type of network. But not calling this when building your JobInfo means the system will assume you do not need any network access and you will not be able to contact your server.
- **Charging and Idle** If your app needs to do work that is resource-heavy, it is highly recommended that you wait until the device is plugged in and/or idle. (Note that "idle" here is not the same as the Doze idle mode, but rather just means the screen is off and it hasn't been used in a little while.)
- **Content Provider update** As of API 24, you can now use a content provider change as a trigger to perform some work. You will need to specify the trigger URI, which will be monitored with a ContentObserver. You can also specify a delay before your job is triggered, if you want to be certain that all changes propagate before your job is run.
- **Backoff criteria** You can specify your own back-off/retry policy. This defaults to an exponential policy, but if you set your own and then return true for rescheduling a job (with onStopJob(), for example), the system will employ your specified policy over the default.
- **Minimum latency and override deadline** If your job cannot start for at least X amount of time, or cannot be delayed past a specific time, you can specify those values here. Even if all conditions have not been met, your job will be run by the deadline (you can check the result

of isOverrideDeadlineExpired() to determine if you're in that case). And if they are met, but your minimum latency has not elapsed, your job will be held.

- **Periodic** If you have work that needs to be done regularly, you can set up a periodic job. This is a great alternative to a repeating alarm for most developers. Because you sort it all out once, schedule it, and the job will run once in each specified period.
- **Persistent** Any work that needs to be persisted across a reboot can be marked as such here. Once the device reboots, the job will be rescheduled according to the conditions. (Note that your app needs the RECEIVE_BOOT_COMPLETED permission for this to work, though.)
- **Extras** If your job needs some information from your app to perform its work, you can pass along primitive data types as extras in the JobInfo.

Finally, consider when to schedule your job.

You can schedule a job using JobScheduler, which you will retrieve from the system. Then, call schedule() using your JobInfo object, and you are good to go. No worries needed.

```
JobScheduler jobScheduler =
    (JobScheduler) getSystemService(Context.JOB_SCHEDULER_SERVICE);
jobScheduler.schedule(new JobInfo.Builder(LOAD_ARTWORK_JOB_ID,
    new ComponentName(this, DownloadArtworkJobService.class))
    .setRequiredNetworkType(JobInfo.NETWORK_TYPE_ANY)
    .build());
```