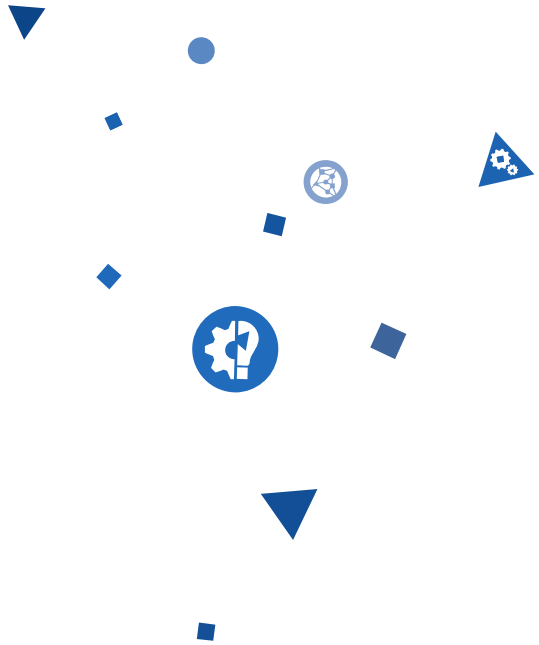

JOBSCHEDULER

JOBSCHEDULER: GENERAL

- 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.
- 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.

JOBSCHEDULER: GENERAL

- 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.

JOBSCHEDULER: HOW TO

- 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.

JOBSCHEDULER: HOW TO

- 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, it will be small, as it simply kicks off something else.

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

JOBSCHEDULER: HOW TO

- 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.
- It requires two parameters:
  - the current job, so that it knows which wakelock can be released
  - 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.

JOBSCHEDULER: HOW TO

- 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.
- 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.

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

JOBSCHEDULER

- Remember: the great advantage to JobScheduler is that it doesn’t perform work solely based on time, but rather based on conditions.
- 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

- Potential criteria you can include in your JobInfo object:
  - NetworkType (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.

JOBSCHEDULER

- Potential criteria you can include in your JobInfo object:
  - 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.

JOBSCHEDULER

- Potential criteria you can include in your JobInfo object:
  - 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. 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.

JOBSCHEDULER

- Potential criteria you can include in your JobInfo object:
  - 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.
  - 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.