

# Android - Oreo's background execution limits & JobScheduler class

With the introduction of Android Oreo (8.0) a new background execution limits announced. In order to save the battery power and reduce RAM usage Google now has major changes in background tasks that apply only when targeting android 26 (the user with android 8.0 can apply these limitations on apps targeting api below 26 in the settings menu).

These limitations are for Services (not Foreground services) and Broadcast Receivers.

## Background Service Limitations

While an app is in the foreground (**either a running activity or a foreground service**), it can create and run both foreground and background services freely. When an app goes into the background, it has a window of several minutes in which it is still allowed to create and use services. At the end of that window, the app is considered to be *idle*. At this time, the system stops the app's background services, just as if the app had called the services' [Service.stopSelf\(\)](#) methods.

Under certain circumstances, a background app is placed on a temporary whitelist for several minutes. While an app is on the whitelist, it can launch services without limitation, and its background services are permitted to run. An app is placed on the whitelist when it handles a task that's visible to the user, such as:

- Handling a high-priority [Firebase Cloud Messaging \(FCM\)](#) message.
- Receiving a broadcast, such as an SMS/MMS message.
- Executing a [PendingIntent](#) from a notification.

Prior to Android 8.0, the usual way to create a foreground service was to create a background service, then promote that service to the foreground. With Android 8.0 the system doesn't allow a background app to create a background service (IllegalStateException will be thrown). For this reason, Android 8.0 introduces the new method [startForegroundService\(\)](#) to start a new service in the foreground. After the system has created the service, the app has five seconds to call the

service's [startForeground\(\)](#) method to show the new service's user-visible notification. If the app does *not* call [startForeground\(\)](#) within the time limit, the system stops the service.

## IntentService

IntentService is a service, and is therefore subject to the new restrictions on background services. As a result, many apps that rely on IntentService do not work properly when targeting Android 8.0 or higher. For this reason, Android Support Library 26.0.0 introduces a new **JobIntentService** class, which provides the same functionality as IntentService but uses jobs instead of services when running on Android 8.0 or higher.

## Broadcast Receivers Limitations

If an app registers to receive broadcasts in her manifest file, each time the broadcast arrives the OS will create an instance of the receiver. This is considered to consume a lot of power in frequent events. For that reason we couldn't register our app in the manifest to ACTION\_SCREEN\_ON or OFF and other frequent broadcasts. Because of that Android Oreo introduced a new Limit:

**Apps that target Android 8.0 or higher can no longer register broadcast receivers for implicit broadcasts in their manifest.** An *implicit broadcast* is a broadcast that does not target that app specifically. For example, [ACTION\\_PACKAGE\\_REPLACED](#) is an implicit broadcast, since it is sent to all registered listeners, letting them know that some package on the device was replaced. However, [ACTION\\_MY\\_PACKAGE\\_REPLACED](#) is not an implicit broadcast, since it is sent only to the app whose package was replaced, no matter how many other apps have registered listeners for that broadcast.

Apps can continue to register for explicit broadcasts in their manifests and can also register receivers dynamically (at run-time) in their Java files.

More than that A number of implicit broadcasts are currently exempted from this limitation. Apps can continue to register receivers for these broadcasts in their manifests, no matter what API level the apps are targeting. List of the exempted broadcasts:

[ACTION\\_LOCKED\\_BOOT\\_COMPLETED](#), [ACTION\\_BOOT\\_COMPLETED](#), [ACTION\\_USER\\_INITIALIZE](#),  
[ACTION\\_TIMEZONE\\_CHANGED](#), [ACTION\\_NEXT\\_ALARM\\_CLOCK\\_CHANGED](#), [ACTION\\_LOCALE\\_CHANGED](#),  
[ACTION\\_USB\\_ACCESSORY\\_ATTACHED](#), [ACTION\\_USB\\_ACCESSORY\\_DETACHED](#),  
[ACTION\\_USB\\_DEVICE\\_ATTACHED](#), [ACTION\\_USB\\_DEVICE\\_DETACHED](#), [ACTION\\_HEADSET\\_PLUG](#),  
[ACTION\\_CONNECTION\\_STATE\\_CHANGED](#), [ACTION\\_ACL\\_CONNECTED](#), [ACTION\\_ACL\\_DISCONNECTED](#),

[ACTION\\_CARRIER\\_CONFIG\\_CHANGED](#), [LOGIN\\_ACCOUNTS\\_CHANGED\\_ACTION](#),  
[ACTION\\_PACKAGE\\_DATA\\_CLEARED](#), [ACTION\\_PACKAGE\\_FULLY\\_REMOVED](#), [ACTION\\_NEW\\_OUTGOING\\_CALL](#),  
[ACTION\\_DEVICE\\_OWNER\\_CHANGED](#), [ACTION\\_EVENT\\_REMINDER](#), [ACTION\\_MEDIA\\_MOUNTED](#),  
[ACTION\\_MEDIA\\_CHECKING](#), [ACTION\\_MEDIA\\_UNMOUNTED](#), [ACTION\\_MEDIA\\_EJECT](#),  
[ACTION\\_MEDIA\\_UNMOUNTABLE](#), [ACTION\\_MEDIA\\_REMOVED](#),  
[ACTION\\_MEDIA\\_BAD\\_REMOVAL](#), [SMS\\_RECEIVED\\_ACTION](#), [WAP\\_PUSH\\_RECEIVED\\_ACTION](#)

## Background Location limits

In an effort to reduce power consumption, Android 8.0 (API level 26) limits how frequently background apps (has no running activity or foreground service) can retrieve the user's current location. Apps can receive location updates only a few times each hour. **These limitations apply to all apps used on devices running Android 8.0 (API level 26) or higher, regardless of an app's target SDK version.**

## Job Scheduler (API 21 and above)

In many cases, your app can replace background services and and implicit broadcast manifest registration with [JobScheduler](#) job.

The main idea in using this API is that Apps can schedule jobs while letting the system optimize based on memory, power, and connectivity conditions. So we can say we can use this API when the job time is not critical for the user. The android system combine jobs together and reduce the battery consumption.

The new job scheduling API differs from the AlarmManager in that it is more aware of the environmental resources and can kick of batch jobs when certain resources are more available.

You use [JobScheduler](#) by registering jobs, specifying their requirements for network and timing. The system then gracefully schedules the jobs to execute at the appropriate times.

To schedule a Job follow these steps:

1. First you need to extends **JobService** base class and declare it in your manifest with the `"android.permission.BIND_JOB_SERVICE" permission`. In the class implement the **onStartJob** and **onStopJob**. If the job fails for some reason, return true from on the **onStopJob** to restart the job. The **onStartJob** is performed in the main thread, if you start asynchronous processing(using **asynctask** or **Thread** instance) in this method, return true otherwise false. Note that unless you create background thread for your job, the work will be done on the Main thread (like **Service**) if you want a background execution queue (similar to **IntentService**) use the new (API 26 - support library) **JobIntentService** and implement the **onHandleWork(Intent)**. All the jobs will enqueued with the **Scheduler's enqueue** function.
2. Then you need to create **JobInfo** object using the **JobInfo.Builder** class. The idea is to specify the scheduling criteria. The job scheduler allows to consider the state of the device, e.g., if it is idle or if network is available at the moment. Configure how the scheduled task should run. You can schedule the task to run under specific conditions, such as:
  - Device is charging
  - Device is connected to an unmetered network
  - Device is idle
  - Start before a certain deadline
  - Start within a predefined time window, e.g., within the next hour
  - Start after a minimal delay, e.g., wait a minimum of 10 minutes

These constraints can be combined. For example, you can schedule a job every 20 minutes, whenever the device is connected to an unmetered network. Deadline is a hard constraint, if that expires the job is always scheduled.

3. Then You will to pass the **JobInfo** to the **JobScheduler** with **schedule(JobInfo)**. Get an instance of the **JobScheduler** with **getSystemService(Context.JOB\_SCHEDULER\_SERVICE)** and call the **schedule()** When the criteria declared are met, the system will execute this job on your application's **JobService** from the first step. If you want the jobs to be executed on a background task

queue use the **JobIntentService** from the first step with the JobScheduler **enqueue** function. Note: The jobInfo you are scheduling Will replace any currently scheduled job with the same **ID** with the new information in the JobInfo. If a job with the given ID is currently running, it will be stopped.

The framework will be intelligent about when you receive your callbacks, and attempt to batch and defer them as much as possible. Typically if you don't specify a deadline on your job, it can be run at any moment depending on the current state of the JobScheduler's internal queue, however it might be deferred as long as until the next time the device is connected to a power source.

For Example:

Lets call the service MyJobService.

```
public class MyJobService extends JobService {  
  
    @Override  
    public boolean onStartJob(JobParameters params) {  
        //do your job asynchronously using AsyncTask  
        return true;  
    }  
  
    @Override  
    public boolean onStopJob(JobParameters params) {  
        return true;  
    }  
}
```

And declare it in your manifest:

```
<service android:name="MyJobService"  
        android:permission="android.permission.BIND_JOB_SERVICE" >
```

Creating the JobInfo:

```
ComponentName serviceComponent = new ComponentName(context, MyJobService.class);  
JobInfo.Builder builder = new JobInfo.Builder(WORK_ID, serviceComponent);  
builder.setMinimumLatency(1 * 1000); // wait at least  
builder.setOverrideDeadline(3 * 1000); // maximum delay  
//builder.setRequiredNetworkType(JobInfo.NETWORK_TYPE_UNMETERED); // require  
unmetered network
```

```
//builder.setRequiresDeviceIdle(true); // device should be idle  
//builder.setRequiresCharging(false); // we don't care if the device is charging or not
```

Get the Job Schedule and schedule the job:

```
JobScheduler jobScheduler =(JobScheduler)getSystemService(JOB_SCHEDULER_SERVICE);
```

And schedule the job:

```
jobScheduler.schedule(builder.build());
```

To read more about Android Oreo API changes please refer to [Android Oreo](#) Android Developers