
QUARTZ

Quartz Scheduler 2.1.x Documentation

Quartz Scheduler 2.1.x Documentation

Quartz Scheduler 2.1.x Documentation

Quartz Scheduler 2.1.x Documentation

Quartz Scheduler 2.1.x Documentation

Quartz Scheduler 2.1.x Documentation

Quartz Overview

Quartz is a full-featured, open source job scheduling service that can be integrated with, or used along side virtually any Java EE or Java SE application - from the smallest stand-alone application to the largest e-commerce system. Quartz can be used to create simple or complex schedules for executing tens, hundreds, or even tens-of-thousands of jobs; jobs whose tasks are defined as standard Java components that are programmed to fulfill the requirements of your application. The Quartz Scheduler includes many enterprise-class features, such as JTA transactions and clustering.

What Can Quartz Do For You?

If your application has tasks that need to occur at given moments in time, or if your system has recurring maintenance jobs then Quartz may be your ideal solution.

Sample uses of job scheduling with Quartz:

- Driving Process Workflow: As a new order is initially placed, schedule a Job to fire in exactly 2 hours, that will check the status of that order, and trigger a warning notification if an order confirmation message has not yet been received for the order, as well as changing the order's status to 'awaiting intervention'.
- System Maintenance: Schedule a job to dump the contents of a database into an XML file every business day (all weekdays except holidays) at 11:30 PM.
- Providing reminder services within an application.

Please refer to our listing of features for more information.

Quartz Features

Runtime Environments

- Quartz can run embedded within another free standing application
- Quartz can be instantiated within an application server (or servlet container), and participate in XA transactions
- Quartz can run as a stand-alone program (within its own Java Virtual Machine), to be used via RMI
- Quartz can be instantiated as a cluster of stand-alone programs (with load-balance and fail-over capabilities)

Job Scheduling

Jobs are scheduled to run when a given Trigger occurs. Triggers can be created with nearly any combination of the following directives:

- at a certain time of day (to the millisecond)
- on certain days of the week
- on certain days of the month
- on certain days of the year
- not on certain days listed within a registered Calendar (such as business holidays)
- repeated a specific number of times
- repeated until a specific time/date
- repeated indefinitely
- repeated with a delay interval

Jobs are given names by their creator and can also be organized into named groups. Triggers may also be given names and placed into groups, in order to easily organize them within the scheduler. Jobs can be added to the scheduler once, but registered with multiple Triggers. Within a J2EE environment, Jobs can perform their work as part of a distributed (XA) transaction.

Job Execution

- Jobs can be any Java class that implements the simple Job interface, leaving infinite possibilities for the work your Jobs can perform.
- Job class instances can be instantiated by Quartz, or by your application's framework.
- When a Trigger occurs, the scheduler notifies zero or more Java objects implementing the JobListener and TriggerListener interfaces (listeners can be simple Java objects, or EJBs, or JMS publishers, etc.). These listeners are also notified after the Job has executed.
- As Jobs are completed, they return a JobCompletionCode which informs the scheduler of success or failure. The JobCompletionCode can also instruct the scheduler of any actions it should take based on the success/fail code - such as immediate re-execution of the Job.

Job Persistence

- The design of Quartz includes a JobStore interface that can be implemented to provide various mechanisms for the storage of jobs.

- With the use of the included JDBCJobStore, all Jobs and Triggers configured as "non-volatile" are stored in a relational database via JDBC.
- With the use of the included RAMJobStore, all Jobs and Triggers are stored in RAM and therefore do not persist between program executions - but this has the advantage of not requiring an external database.

Transactions

- Quartz can participate in JTA transactions, via the use of JobStoreCMT (a subclass of JDBCJobStore).
- Quartz can manage JTA transactions (begin and commit them) around the execution of a Job, so that the work performed by the Job automatically happens within a JTA transaction.

Clustering

- Fail-over.
- Load balancing.
- Quartz's built-in clustering features rely upon database persistence via JDBCJobStore (described above).
- Terracotta extensions to Quartz provide clustering capabilities without the need for a backing database.

Listeners & Plug-Ins

- Applications can catch scheduling events to monitor or control job/trigger behavior by implementing one or more listener interfaces.
- The Plug-In mechanism can be used add functionality to Quartz, such keeping a history of job executions, or loading job and trigger definitions from a file.
- Quartz ships with a number of "factory built" plug-ins and listeners.

What's New In Quartz Scheduler 2.1

If you aren't yet familiar with Quartz 2.0, you may want to first read What's New In Quartz 2.0.

We'd like to express thanks to the community contributors that performed a significant amount of the work contained in this release!

API Changes

- QTZ-197 - JobDataMap has had improvements made to its interface w/respect to generics
- QTZ-184 - GroupMatcher API changes to avoid generics compiler warnings

New Features

- QTZ-196 - New trigger type 'DailyTimeIntervalTrigger'
- QTZ-186 - Improvements for interrupting executing jobs

Miscellaneous

- Performance improvements, including:
    - *Now Implemented In JDBC-JobStore*: Ability to batch-acquire triggers that are ready to be fired, which can provide performance improvements for very busy schedulers (TerracottaJobStore and RAMJobStore got this feature with Quartz 2.0). NOTE: If "org.quartz.scheduler.batchTriggerAcquisitionMaxCount" is set to > 1, and JDBC JobStore is used, then "org.quartz.jobStore.acquireTriggersWithinLock" must be set to "true" to avoid data corruption.
- PropertySettingJobFactory is now the default JobFactory.
- Various bug fixes, for complete listing see the release notes from Jira: https://jira.terracotta.org/jira/secure/ReleaseNote.jspa?projectId=10282&version=10981

Quartz Quick Start Guide

(Primarily authored by Dafydd James)

Welcome to the QuickStart guide for Quartz. As you read this guide, expect to see details of:

- Downloading Quartz
- Installing Quartz
- Configuring Quartz to your own particular needs
- Starting a sample application

After becoming familiar with the basic functioning of Quartz Scheduler, consider more advanced features such as Where, an Enterprise feature that allows jobs and triggers to run on specified Terracotta clients instead of randomly chosen
ones.

Download and Install

First, Download the most recent stable release - registration is not required. Unpack the distribution and install it so that your application can see it.

The Quartz JAR Files

The Quartz package includes a number of jar files, located in root directory of the distribution. The main Quartz library is named quartz-all-xxx.jar (where xxx is a version number). In order to use any of Quartz's features, this jar must be located on your application's classpath.

Once you've downloaded Quartz, unzip it somewhere, grab the quartz-all-xxx.jar and put it where you want it. (If you need information on how to unzip files, go away and learn before you go anywhere near a development environment or the Internet in general. Seriously.)

I use Quartz primarily within an application server environment, so my preference is to include the Quartz JAR within my enterprise application (.ear or .war file). However, if you want to make Quartz available to many applications then simply make sure it's on the classpath of your appserver. If you are making a stand-alone application, place it on the application's classpath with all of the other JARs your application depends upon.

Quartz depends on a number of third-party libraries (in the form of jars) which are included in the distribution .zip file in the 'lib' directory. To use all the features of Quartz, all of these jars must also exist on your classpath. If you're building a stand-alone Quartz application, I suggest you simply add all of them to the classpath. If you're using Quartz within an app server environment, at least some of the jars will likely already exist on the classpath, so you can afford (if you want) to be a bit more selective as to which jars you include.

> In an appserver environment, beware of strange results when accidentally including two different versions of the same jar. For example, WebLogic includes an implementation of J2EE (inside weblogic.jar) which may differ to the one in servlet.jar. In this case, it's usually better to leave servlet.jar out of your application, so you know which classes are being utilized.

The Properties File

Quartz uses a properties file called (kudos on the originality) quartz.properties. This isn't necessary at first, but to use anything but the most basic configuration it must be located on your classpath.

Again, to give an example based on my personal situation, my application was developed using WebLogic Workshop. I keep all of my configuration files (including quartz.properties) in a project under the root of my application. When I package everything up into a .ear file, the config project gets packaged into a .jar which is included within the final .ear. This automatically puts quartz.properties on the classpath.

If you're building a web application (i.e. in the form of a .war file) that includes Quartz, you will likely want to place the quartz.properties file in the WEB-INF/classes folder in order for it to be on the classpath.

Configuration

This is the big bit! Quartz is a very configurable application. The best way to configure Quartz is to edit a quartz.properties file, and place it in your application's classpath (see Installation section above).

There are several example properties files that ship within the Quartz distribution, particularly under the examples/ directory. I would suggest you create your own quartz.properties file, rather than making a copy of one of the examples and deleting the bits you don't need. It's neater that way, and you'll explore more of what Quartz has to offer.

Full documentation of available properties is available in the Quartz Configuration Reference.

To get up and running quickly, a basic quartz.properties looks something like this:

```
org.quartz.scheduler.instanceName = MyScheduler
org.quartz.threadPool.threadCount = 3
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

The scheduler created by this configuration has the following characteristics:

- org.quartz.scheduler.instanceName - This scheduler's name will be "MyScheduler".
- org.quartz.threadPool.threadCount - There are 3 threads in the thread pool, which means that a maximum of 3 jobs can be run simultaneously.
- org.quartz.jobStore.class - All of Quartz's data, such as details of jobs and triggers, is held in memory (rather than in a database). Even if you have a database and want to use it with Quartz, I suggest you get Quartz working with the RamJobStore before you open up a whole new dimension by working with a database.

Starting a Sample Application

Now you've downloaded and installed Quartz, it's time to get a sample application up and running. The following code obtains an instance of the scheduler, starts it, then shuts it down:

QuartzTest.java

```
import org.quartz.Scheduler;
import org.quartz.SchedulerException;
import org.quartz.impl.StdSchedulerFactory;
```

```
import static org.quartz.JobBuilder.*;
import static org.quartz.TriggerBuilder.*;
import static org.quartz.SimpleScheduleBuilder.*;


public class QuartzTest {

    public static void main(String[] args) {

        try {
            // Grab the Scheduler instance from the Factory
            Scheduler scheduler = StdSchedulerFactory.getDefaultScheduler();

            // and start it off
            scheduler.start();

            scheduler.shutdown();

        } catch (SchedulerException se) {
            se.printStackTrace();
        }
    }
}
```

> Once you obtain a scheduler using StdSchedulerFactory.getDefaultScheduler(), your application will not terminate until you call scheduler.shutdown(), because there will be active threads.

Note the static imports in the code example; these will come into play in the code example below.

If you have not set up logging, all logs will be sent to the console and your output will look something like this:

```
[INFO] 21 Jan 08:46:27.857 AM main [org.quartz.core.QuartzScheduler]
Quartz Scheduler v.2.0.0-SNAPSHOT created.

[INFO] 21 Jan 08:46:27.859 AM main [org.quartz.simpl.RAMJobStore]
RAMJobStore initialized.

[INFO] 21 Jan 08:46:27.865 AM main [org.quartz.core.QuartzScheduler]
Scheduler meta-data: Quartz Scheduler (v2.0.0) 'Scheduler' with instanceId 'NON_CLUSTERED'
  Scheduler class: 'org.quartz.core.QuartzScheduler' - running locally.
  NOT STARTED.
  Currently in standby mode.
  Number of jobs executed: 0
  Using thread pool 'org.quartz.simpl.SimpleThreadPool' - with 50 threads.
  Using job-store 'org.quartz.simpl.RAMJobStore' - which does not support persistence. and is not


[INFO] 21 Jan 08:46:27.865 AM main [org.quartz.impl.StdSchedulerFactory]
Quartz scheduler 'Scheduler' initialized from default resource file in Quartz package: 'quartz.pr

[INFO] 21 Jan 08:46:27.866 AM main [org.quartz.impl.StdSchedulerFactory]
Quartz scheduler version: 2.0.0

[INFO] 21 Jan 08:46:27.866 AM main [org.quartz.core.QuartzScheduler]
Scheduler Scheduler_$_NON_CLUSTERED started.

[INFO] 21 Jan 08:46:27.866 AM main [org.quartz.core.QuartzScheduler]
Scheduler Scheduler_$_NON_CLUSTERED shutting down.
```

```
[INFO] 21 Jan 08:46:27.866 AM main [org.quartz.core.QuartzScheduler]
Scheduler Scheduler_$_NON_CLUSTERED paused.

[INFO] 21 Jan 08:46:27.867 AM main [org.quartz.core.QuartzScheduler]
Scheduler Scheduler_$_NON_CLUSTERED shutdown complete.
```

To do something interesting, you need code between the *start()* and *shutdown()* calls.

```
   // define the job and tie it to our HelloJob class
    JobDetail job = newJob(HelloJob.class)
        .withIdentity("job1", "group1")
        .build();

    // Trigger the job to run now, and then repeat every 40 seconds
    Trigger trigger = newTrigger()
        .withIdentity("trigger1", "group1")
        .startNow()
        .withSchedule(simpleSchedule()
                .withIntervalInSeconds(40)
                .repeatForever())
        .build();

    // Tell quartz to schedule the job using our trigger
    scheduler.scheduleJob(job, trigger);
```

(you will also need to allow some time for the job to be triggered and executed before calling shutdown() - for a simple example such as this, you might just want to add a *Thread.sleep(60000)* call).

Now go have some fun!

Quartz Enterprise Job Scheduler Tutorials

Before starting the tutorial, you may first want to review the Quick Start Guide, which covers download, installation, and very basic configuration of Quartz.

Choose a Lesson:

Choose a Special Topic:

Table of Contents | Lesson 2 ›

Tutorial

Lesson 1: Using Quartz

Before you can use the scheduler, it needs to be instantiated (who'd have guessed?). To do this, you use a SchedulerFactory. Some users of Quartz may keep an instance of a factory in a JNDI store, others may find it just as easy (or easier) to instantiate and use a factory instance directly (such as in the example below).

Once a scheduler is instantiated, it can be started, placed in stand-by mode, and shutdown. Note that once a scheduler is shutdown, it cannot be restarted without being re-instantiated. Triggers do not fire (jobs do not execute) until the scheduler has been started, nor while it is in the paused state.

Here's a quick snippet of code, that instantiates and starts a scheduler, and schedules a job for execution:

```
SchedulerFactory schedFact = new org.quartz.impl.StdSchedulerFactory();

Scheduler sched = schedFact.getScheduler();

sched.start();

// define the job and tie it to our HelloJob class
JobDetail job = newJob(HelloJob.class)
    .withIdentity("myJob", "group1")
    .build();

// Trigger the job to run now, and then every 40 seconds
Trigger trigger = newTrigger()
    .withIdentity("myTrigger", "group1")
    .startNow()
    .withSchedule(simpleSchedule()
        .withIntervalInSeconds(40)
        .repeatForever())
    .build();

// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger);
```

As you can see, working with quartz is rather simple. In Lesson 2 we'll give a quick overview of Jobs and Triggers, and Quartz's API so that you can more fully understand this example.

Table of Contents | Lesson 3 ›

Tutorial

Lesson 2: The Quartz API, Jobs And Triggers

The Quartz API

The key interfaces of the Quartz API are:

- Scheduler - the main API for interacting with the scheduler.
- Job - an interface to be implemented by components that you wish to have executed by the scheduler.
- JobDetail - used to define instances of Jobs.
- Trigger - a component that defines the schedule upon which a given Job will be executed.
- JobBuilder - used to define/build JobDetail instances, which define instances of Jobs.
- TriggerBuilder - used to define/build Trigger instances.

A **Scheduler**'s life-cycle is bounded by it's creation, via a **SchedulerFactory** and a call to its *shutdown()* method. Once created the Scheduler interface can be used add, remove, and list Jobs and Triggers, and perform other scheduling-related operations (such as pausing a trigger). However, the Scheduler will not actually act on any triggers (execute jobs) until it has been started with the *start()* method, as shown in Lesson 1.

Quartz provides "builder" classes that define a Domain Specific Language (or DSL, also sometimes referred to as a "fluent interface"). In the previous lesson you saw an example of it, which we present a portion of here again:

```
// define the job and tie it to our HelloJob class
JobDetail job = newJob(HelloJob.class)
    .withIdentity("myJob", "group1") // name "myJob", group "group1"
    .build();

// Trigger the job to run now, and then every 40 seconds
Trigger trigger = newTrigger()
    .withIdentity("myTrigger", "group1")
    .startNow()
    .withSchedule(simpleSchedule()
        .withIntervalInSeconds(40)
        .repeatForever())
    .build();

// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger);
```

The block of code that builds the job definition is using methods that were statically imported from the ***JobBuilder*** class. Likewise, the block of code that builds the trigger is using methods imported from the ***TriggerBuilder*** class - as well as from the ***SimpleScheduleBulder*** class.

The static imports of the DSL can be achieved through import statements such as these:

```
import static org.quartz.JobBuilder.*;
import static org.quartz.SimpleScheduleBuilder.*;
import static org.quartz.CronScheduleBuilder.*;
import static org.quartz.CalendarIntervalScheduleBuilder.*;
import static org.quartz.TriggerBuilder.*;
```

```
import static org.quartz.DateBuilder.*;
```

The various "*ScheduleBuilder*" classes have methods relating to defining different types of schedules.

The *DateBuilder* class contains various methods for easily constructing *java.util.Date* instances for particular points in time (such as a date that represents the next even hour - or in other words 10:00:00 if it is currently 9:43:27).

Jobs and Triggers

A Job is a class that implements the ***Job*** interface, which has only one simple method:

The Job Interface

```
package org.quartz;

public interface Job {

  public void execute(JobExecutionContext context)
    throws JobExecutionException;
}
```

When the Job's trigger fires (more on that in a moment), the execute(..) method is invoked by one of the scheduler's worker threads. The *JobExecutionContext* object that is passed to this method provides the job instance with information about its "run-time" environment - a handle to the Scheduler that executed it, a handle to the Trigger that triggered the execution, the job's JobDetail object, and a few other items.

The *JobDetail* object is created by the Quartz client (your program) at the time the Job is added to the scheduler. It contains various property settings for the Job, as well as a *JobDataMap*, which can be used to store state information for a given instance of your job class. It is essentially the definition of the job instance, and is discussed in further detail in the next lesson.

Trigger objects are used to trigger the execution (or 'firing') of jobs. When you wish to schedule a job, you instantiate a trigger and 'tune' its properties to provide the scheduling you wish to have. Triggers may also have a JobDataMap associated with them - this is useful to passing parameters to a Job that are specific to the firings of the trigger. Quartz ships with a handful of different trigger types, but the most commonly used types are SimpleTrigger and CronTrigger.

SimpleTrigger is handy if you need 'one-shot' execution (just single execution of a job at a given moment in time), or if you need to fire a job at a given time, and have it repeat N times, with a delay of T between executions. CronTrigger is useful if you wish to have triggering based on calendar-like schedules - such as "every Friday, at noon" or "at 10:15 on the 10th day of every month."

Why Jobs AND Triggers? Many job schedulers do not have separate notions of jobs and triggers. Some define a 'job' as simply an execution time (or schedule) along with some small job identifier. Others are much like the union of Quartz's job and trigger objects. While developing Quartz, we decided that it made sense to create a separation between the schedule and the work to be performed on that schedule. This has (in our opinion) many benefits.

For example, Jobs can be created and stored in the job scheduler independent of a trigger, and many triggers can be associated with the same job. Another benefit of this loose-coupling is the ability to configure jobs that remain in the scheduler after their associated triggers have expired, so that that it can be rescheduled later,

without having to re-define it. It also allows you to modify or replace a trigger without having to re-define its associated job.

Identities

Jobs and Triggers are given identifying keys as they are registered with the Quartz scheduler. The keys of Jobs and Triggers (JobKey and TriggerKey) allow them to be placed into 'groups' which can be useful for organizing your jobs and triggers into categories such as "reporting jobs" and "maintenance jobs". The name portion of the key of a job or trigger must be unique within the group - or in other words, the complete key (or identifier) of a job or trigger is the compound of the name and group.

You now have a general idea about what Jobs and Triggers are, you can learn more about them in Lesson 3: More About Jobs & JobDetails and Lesson 4: More About Triggers.

Table of Contents | ‹ Lesson 2 | Lesson 4 ›

Tutorial

Lesson 3: More About Jobs and Job Details

As you saw in Lesson 2, Jobs are rather easy to implement, having just a single 'execute' method in the interface. There are just a few more things that you need to understand about the nature of jobs, about the execute(..) method of the Job interface, and about JobDetails.

While a job class that you implement has the code that knows how do do the actual work of the particular type of job, Quartz needs to be informed about various attributes that you may wish an instance of that job to have. This is done via the JobDetail class, which was mentioned briefly in the previous section.

JobDetail instances are built using the JobBuilder class. You will typically want to use a static import of all of its methods, in order to have the DSL-feel within your code.

```
import static org.quartz.JobBuilder.*;
```

Let's take a moment now to discuss a bit about the 'nature' of Jobs and the life-cycle of job instances within Quartz. First lets take a look back at some of that snippet of code we saw in Lesson 1:

```
// define the job and tie it to our HelloJob class
JobDetail job = newJob(HelloJob.class)
    .withIdentity("myJob", "group1") // name "myJob", group "group1"
    .build();

// Trigger the job to run now, and then every 40 seconds
Trigger trigger = newTrigger()
    .withIdentity("myTrigger", "group1")
    .startNow()
    .withSchedule(simpleSchedule()
        .withIntervalInSeconds(40)
        .repeatForever())
    .build();

// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger);
```

Now consider the job class "HelloJob" defined as such:

```
public class HelloJob implements Job {

  public HelloJob() {
  }

  public void execute(JobExecutionContext context)
    throws JobExecutionException
  {
    System.err.println("Hello!  HelloJob is executing.");
  }
}
```

Notice that we give the scheduler a JobDetail instance, and that it knows the type of job to be executed by simply providing the job's class as we build the JobDetail. Each (and every) time the scheduler executes the job, it creates a new instance of the class before calling its execute(..) method. When the execution is

complete, references to the job class instance are dropped, and the instance is then garbage collected. One of the ramifications of this behavior is the fact that jobs must have a no-argument constructor (when using the default JobFactory implementation). Another ramification is that it does not make sense to have state data-fields defined on the job class - as their values would not be preserved between job executions.

You may now be wanting to ask "how can I provide properties/configuration for a Job instance?" and "how can I keep track of a job's state between executions?" The answer to these questions are the same: the key is the JobDataMap, which is part of the JobDetail object.

JobDataMap

The JobDataMap can be used to hold any amount of (serializable) data objects which you wish to have made available to the job instance when it executes. JobDataMap is an implementation of the Java Map interface, and has some added convenience methods for storing and retrieving data of primitive types.

Here's some quick snippets of putting data into the JobDataMap while defining/building the JobDetail, prior to adding the job to the scheduler:

```
// define the job and tie it to our DumbJob class
JobDetail job = newJob(DumbJob.class)
    .withIdentity("myJob", "group1") // name "myJob", group "group1"
    .usingJobData("jobSays", "Hello World!")
    .usingJobData("myFloatValue", 3.141f)
    .build();
```

Here's a quick example of getting data from the JobDataMap during the job's execution:

```
public class DumbJob implements Job {

    public DumbJob() {
    }

    public void execute(JobExecutionContext context)
      throws JobExecutionException
    {
      JobKey key = context.getJobDetail().getKey();

      JobDataMap dataMap = context.getJobDetail().getJobDataMap();

      String jobSays = dataMap.getString("jobSays");
      float myFloatValue = dataMap.getFloat("myFloatValue");

      System.err.println("Instance " + key + " of DumbJob says: " + jobSays + ", and val is: " +
    }
}
```

If you use a persistent JobStore (discussed in the JobStore section of this tutorial) you should use some care in deciding what you place in the JobDataMap, because the object in it will be serialized, and they therefore become prone to class-versioning problems. Obviously standard Java types should be very safe, but beyond that, any time someone changes the definition of a class for which you have serialized instances, care has to be taken not to break compatibility. Further information on this topic can be found in this Java Developer Connection Tech Tip: Serialization In The Real World. Optionally, you can put JDBC-JobStore and JobDataMap into a mode where only primitives and strings are allowed to be stored in the map, thus eliminating any possibility of later serialization problems.

If you add setter methods to your job class that correspond to the names of keys in the JobDataMap (such as a *setJobSays(String val)* method for the data in the example above), then Quartz's default JobFactory implementation will automatically call those setters when the job is instantiated, thus preventing the need to explicitly get the values out of the map within your execute method.

Triggers can also have JobDataMaps associated with them. This can be useful in the case where you have a Job that is stored in the scheduler for regular/repeated use by multiple Triggers, yet with each independent triggering, you want to supply the Job with different data inputs.

The JobDataMap that is found on the JobExecutionContext during Job execution serves as a convenience. It is a merge of the JobDataMap found on the JobDetail and the one found on the Trigger, with the values in the latter overriding any same-named values in the former.

Here's a quick example of getting data from the JobExecutionContext's merged JobDataMap during the job's execution:

```
public class DumbJob implements Job {

    public DumbJob() {
    }

    public void execute(JobExecutionContext context)
      throws JobExecutionException
    {
      JobKey key = context.getJobDetail().getKey();

      JobDataMap dataMap = context.getMergedJobDataMap();  // Note the difference from the previo

      String jobSays = dataMap.getString("jobSays");
      float myFloatValue = dataMap.getFloat("myFloatValue");
      ArrayList state = (ArrayList)dataMap.get("myStateData");
      state.add(new Date());

      System.err.println("Instance " + key + " of DumbJob says: " + jobSays + ", and val is: " +
    }
  }
```

Or if you wish to rely on the JobFactory "injecting" the data map values onto your class, it might look like this instead:

```
public class DumbJob implements Job {


    String jobSays;
    float myFloatValue;
    ArrayList state;

    public DumbJob() {
    }

    public void execute(JobExecutionContext context)
      throws JobExecutionException
    {
      JobKey key = context.getJobDetail().getKey();

      JobDataMap dataMap = context.getMergedJobDataMap();  // Note the difference from the previo

      state.add(new Date());
```

```
      System.err.println("Instance " + key + " of DumbJob says: " + jobSays + ", and val is: " +
    }

    public void setJobSays(String jobSays) {
      this.jobSays = jobSays;
    }

    public void setMyFloatValue(float myFloatValue) {
      myFloatValue = myFloatValue;
    }

    public void setState(ArrayList state) {
      state = state;
    }

  }
```

You'll notice that the overall code of the class is longer, but the code in the execute() method is cleaner. One could also argue that although the code is longer, that it actually took less coding, if the programmer's IDE was used to auto-generate the setter methods, rather than having to hand-code the individual calls to retrieve the values from the JobDataMap. The choice is yours.

Job "Instances"

Many users spend time being confused about what exactly constitutes a "job instance". We'll try to clear that up here and in the section below about job state and concurrency.

You can create a single job class, and store many 'instance definitions' of it within the scheduler by creating multiple instances of JobDetails - each with its own set of properties and JobDataMap - and adding them all to the scheduler.

For example, you can create a class that implements the Job interface called "SalesReportJob". The job might be coded to expect parameters sent to it (via the JobDataMap) to specify the name of the sales person that the sales report should be based on. They may then create multiple definitions (JobDetails) of the job, such as "SalesReportForJoe" and "SalesReportForMike" which have "joe" and "mike" specified in the corresponding JobDataMaps as input to the respective jobs.

When a trigger fires, the JobDetail (instance definition) it is associated to is loaded, and the job class it refers to is instantiated via the JobFactory configured on the Scheduler. The default JobFactory simply calls newInstance() on the job class, then attempts to call setter methods on the class that match the names of keys within the JobDataMap. You may want to create your own implementation of JobFactory to accomplish things such as having your application's IoC or DI container produce/initialize the job instance.

In "Quartz speak", we refer to each stored JobDetail as a "job definition" or "JobDetail instance", and we refer to a each executing job as a "job instance" or "instance of a job definition". Usually if we just use the word "job" we are referring to a named definition, or JobDetail. When we are referring to the class implementing the job interface, we usually use the term "job class".

Job State and Concurrency

Now, some additional notes about a job's state data (aka JobDataMap) and concurrency. There are a couple annotations that can be added to your Job class that affect Quartz's behavior with respect to these aspects.

@DisallowConcurrentExecution is an annotation that can be added to the Job class that tells Quartz not to execute multiple instances of a given job definition (that refers to the given job class) concurrently. Notice the wording there, as it was chosen very carefully. In the example from the previous section, if "SalesReportJob" has this annotation, than only one instance of "SalesReportForJoe" can execute at a given time, but it *can* execute concurrently with an instance of "SalesReportForMike". The constraint is based upon an instance definition (JobDetail), not on instances of the job class. However, it was decided (during the design of Quartz) to have the annotation carried on the class itself, because it does often make a difference to how the class is coded.

@PersistJobDataAfterExecution is an annotation that can be added to the Job class that tells Quartz to update the stored copy of the JobDetail's JobDataMap after the execute() method completes successfully (without throwing an exception), such that the next execution of the same job (JobDetail) receives the updated values rather than the originally stored values. Like the *@DisallowConcurrentExecution* annotation, this applies to a job definition instance, not a job class instance, though it was decided to have the job class carry the attribute because it does often make a difference to how the class is coded (e.g. the 'statefulness' will need to be explicitly 'understood' by the code within the execute method).

If you use the *@PersistJobDataAfterExecution* annotation, you should strongly consider also using the *@DisallowConcurrentExecution* annotation, in order to avoid possible confusion (race conditions) of what data was left stored when two instances of the same job (JobDetail) executed concurrently.

Other Attributes Of Jobs

Here's a quick summary of the other properties which can be defined for a job instance via the JobDetail object:

- Durability - if a job is non-durable, it is automatically deleted from the scheduler once there are no longer any active triggers associated with it. In other words, non-durable jobs have a life span bounded by the existence of its triggers.
- RequestsRecovery - if a job "requests recovery", and it is executing during the time of a 'hard shutdown' of the scheduler (i.e. the process it is running within crashes, or the machine is shut off), then it is re-executed when the scheduler is started again. In this case, the JobExecutionContext.isRecovering() method will return true.

JobExecutionException

Finally, we need to inform you of a few details of the `Job.execute(..)` method. The only type of exception (including RuntimeExceptions) that you are allowed to throw from the execute method is the JobExecutionException. Because of this, you should generally wrap the entire contents of the execute method with a 'try-catch' block. You should also spend some time looking at the documentation for the JobExecutionException, as your job can use it to provide the scheduler various directives as to how you want the exception to be handled.

Table of Contents | ‹ Lesson 3 | Lesson 5 ›

Tutorial

Lesson 4: More About Triggers

Like jobs, triggers are quite easy to work with, but do contain a variety of customizable options that you need to be aware of and understand before you can make full use of Quartz. Also, as noted earlier, there are different types of triggers that you can select from to meet different scheduling needs.

You will learn about the two most common types of triggers in Lesson 5: Simple Triggers and Lesson 6: Cron Triggers.

Common Trigger Attributes

Aside from the fact that all trigger types have TriggerKey properties for tracking their identities, there are a number of other properties that are common to all trigger types. These common properties are set using the TriggerBuilder when you are building the trigger definition (examples of that will follow).

Here is a listing of properties common to all trigger types:

- The "jobKey" property indicates the identity of the job that should be executed when the trigger fires.
- The "startTime" property indicates when the trigger's schedule first comes into affect. The value is a *java.util.Date* object that defines a moment in time on a given calendar date. For some trigger types, the trigger will actually fire at the start time, for others it simply marks the time that the schedule should start being followed. This means you can store a trigger with a schedule such as "every 5th day of the month" during January, and if the startTime property is set to April 1st, it will be a few months before the first firing.
- The "endTime" property indicates when the trigger's schedule should no longer be in effect. In other words, a trigger with a schedule of "every 5th day of the month" and with an end time of July 1st will fire for it's last time on June 5th.

Other properties, which take a bit more explanation are discussed in the following sub-sections.

Priority

Sometimes, when you have many Triggers (or few worker threads in your Quartz thread pool), Quartz may not have enough resources to immediately fire all of the Triggers that are scheduled to fire at the same time. In this case, you may want to control which of your Triggers get first crack at the available Quartz worker threads. For this purpose, you can set the *priority* property on a Trigger. If N Triggers are to fire at the same time, but there are only Z worker threads currently available, then the first Z Triggers with the *highest* priority will be executed first. If you do not set a priority on a Trigger, then it will use the default priority of 5. Any integer value is allowed for priority, positive or negative.

Note: Priorities are only compared when triggers have the same fire time. A trigger scheduled to fire at 10:59 will always fire before one scheduled to fire at 11:00.

Note: When a trigger's job is detected to require recovery, its recovery is scheduled with the same priority as the original trigger.

Misfire Instructions

Another important property of a Trigger is its "misfire instruction". A misfire occurs if a persistent trigger "misses" its firing time because of the scheduler being shutdown, or because there are no available threads in Quartz's thread pool for executing the job. The different trigger types have different misfire instructions available to them. By default they use a 'smart policy' instruction - which has dynamic behavior based on trigger type and configuration. When the scheduler starts, it searches for any persistent triggers that have misfired, and it then updates each of them based on their individually configured misfire instructions. When you start using Quartz in your own projects, you should make yourself familiar with the misfire instructions that are defined on the given trigger types, and explained in their JavaDoc. More specific information about misfire instructions will be given within the tutorial lessons specific to each trigger type.

Calendars

Quartz ***Calendar*** objects (not java.util.Calendar objects) can be associated with triggers at the time the trigger is defined and stored in the scheduler. Calendars are useful for excluding blocks of time from the the trigger's firing schedule. For instance, you could create a trigger that fires a job every weekday at 9:30 am, but then add a Calendar that excludes all of the business's holidays.

Calendar's can be any serializable objects that implement the Calendar interface, which looks like this:

The Calendar Interface

```
package org.quartz;

public interface Calendar {

  public boolean isTimeIncluded(long timeStamp);

  public long getNextIncludedTime(long timeStamp);

}
```

Notice that the parameters to these methods are of the long type. As you may guess, they are timestamps in millisecond format. This means that calendars can 'block out' sections of time as narrow as a millisecond. Most likely, you'll be interested in 'blocking-out' entire days. As a convenience, Quartz includes the class org.quartz.impl.HolidayCalendar, which does just that.

Calendars must be instantiated and registered with the scheduler via the addCalendar(..) method. If you use HolidayCalendar, after instantiating it, you should use its addExcludedDate(Date date) method in order to populate it with the days you wish to have excluded from scheduling. The same calendar instance can be used with multiple triggers such as this:

Calendar Example

```
HolidayCalendar cal = new HolidayCalendar();
cal.addExcludedDate( someDate );
cal.addExcludedDate( someOtherDate );

sched.addCalendar("myHolidays", cal, false);


Trigger t = newTrigger()
    .withIdentity("myTrigger")
```

```
    .forJob("myJob")
    .withSchedule(dailyAtHourAndMinute(9, 30)) // execute job daily at 9:30
    .modifiedByCalendar("myHolidays") // but not on holidays
    .build();

// .. schedule job with trigger

Trigger t2 = newTrigger()
    .withIdentity("myTrigger2")
    .forJob("myJob2")
    .withSchedule(dailyAtHourAndMinute(11, 30)) // execute job daily at 11:30
    .modifiedByCalendar("myHolidays") // but not on holidays
    .build();

// .. schedule job with trigger2
```

The details of the construction/building of triggers will be given in the next couple lessons. For now, just believe that the code above creates two triggers, each scheduled to fire daily. However, any of the firings that would have occurred during the period excluded by the calendar will be skipped.

See the *org.quartz.impl.calendar* package for a number of Calendar implementations that may suit your needs.

Table of Contents | ‹ Lesson 4 | Lesson 6 ›

Tutorial

Lesson 5: SimpleTrigger

SimpleTrigger should meet your scheduling needs if you need to have a job execute exactly once at a specific moment in time, or at a specific moment in time followed by repeats at a specific interval. For example, if you want the trigger to fire at exactly 11:23:54 AM on January 13, 2015, or if you want it to fire at that time, and then fire five more times, every ten seconds.

With this description, you may not find it surprising to find that the properties of a SimpleTrigger include: a start-time, and end-time, a repeat count, and a repeat interval. All of these properties are exactly what you'd expect them to be, with only a couple special notes related to the end-time property.

The repeat count can be zero, a positive integer, or the constant value SimpleTrigger.REPEAT_INDEFINITELY. The repeat interval property must be zero, or a positive long value, and represents a number of milliseconds. Note that a repeat interval of zero will cause 'repeat count' firings of the trigger to happen concurrently (or as close to concurrently as the scheduler can manage).

If you're not already familiar with Quartz's DateBuilder class, you may find it helpful for computing your trigger fire-times, depending on the ***startTime*** (or endTime) that you're trying to create.

The ***endTime*** property (if it is specified) overrides the repeat count property. This can be useful if you wish to create a trigger such as one that fires every 10 seconds until a given moment in time - rather than having to compute the number of times it would repeat between the start-time and the end-time, you can simply specify the end-time and then use a repeat count of REPEAT_INDEFINITELY (you could even specify a repeat count of some huge number that is sure to be more than the number of times the trigger will actually fire before the end-time arrives).

SimpleTrigger instances are built using TriggerBuilder (for the trigger's main properties) and SimpleScheduleBuilder (for the SimpleTrigger-specific properties). To use these builders in a DSL-style, use static imports:

```
import static org.quartz.TriggerBuilder.*;
import static org.quartz.SimpleScheduleBuilder.*;
import static org.quartz.DateBuilder.*:
```

Here are various examples of defining triggers with simple schedules, read through them all, as they each show at least one new/different point:

Build a trigger for a specific moment in time, with no repeats:

```
SimpleTrigger trigger = (SimpleTrigger) newTrigger()
  .withIdentity("trigger1", "group1")
  .startAt(myStartTime) // some Date
  .forJob("job1", "group1") // identify job with name, group strings
  .build();
```

Build a trigger for a specific moment in time, then repeating every ten seconds ten times:

```
trigger = newTrigger()
  .withIdentity("trigger3", "group1")
```

```
    .startAt(myTimeToStartFiring)  // if a start time is not given (if this line were omitted), "
    .withSchedule(simpleSchedule()
        .withIntervalInSeconds(10)
        .withRepeatCount(10)) // note that 10 repeats will give a total of 11 firings
    .forJob(myJob) // identify job with handle to its JobDetail itself
    .build();
```

Build a trigger that will fire once, five minutes in the future:

```
trigger = (SimpleTrigger) newTrigger()
  .withIdentity("trigger5", "group1")
  .startAt(futureDate(5, IntervalUnit.MINUTE)) // use DateBuilder to create a date in the futur
  .forJob(myJobKey) // identify job with its JobKey
  .build();
```

Build a trigger that will fire now, then repeat every five minutes, until the hour 22:00:

```
trigger = newTrigger()
  .withIdentity("trigger7", "group1")
  .withSchedule(simpleSchedule()
      .withIntervalInMinutes(5)
      .repeatForever())
  .endAt(dateOf(22, 0, 0))
  .build();
```

Build a trigger that will fire at the top of the next hour, then repeat every 2 hours, forever:

```
trigger = newTrigger()
  .withIdentity("trigger8") // because group is not specified, "trigger8" will be in the defaul
  .startAt(evenHourDate(null)) // get the next even-hour (minutes and seconds zero ("00:00"))
  .withSchedule(simpleSchedule()
      .withIntervalInHours(2)
      .repeatForever())
  // note that in this example, 'forJob(..)' is not called
  //  - which is valid if the trigger is passed to the scheduler along with the job
  .build();

  scheduler.scheduleJob(trigger, job);
```

Spend some time looking at all of the available methods in the language defined by TriggerBuilder and SimpleScheduleBuilder so that you can be familiar with options available to you that may not have been demonstrated in the examples above.

> Note that TriggerBuilder (and Quartz's other builders) will generally choose a reasonable value for properties that you do not explicitly set. For examples: if you don't call one of the *withIdentity(..)* methods, then TriggerBuilder will generate a random name for your trigger; if you don't call *startAt(..)* then the current time (immediately) is assumed.

SimpleTrigger Misfire Instructions

SimpleTrigger has several instructions that can be used to inform Quartz what it should do when a misfire occurs. (Misfire situations were introduced in "Lesson 4: More About Triggers"). These instructions are defined as constants on SimpleTrigger itself (including JavaDoc describing their behavior). The instructions include:

Misfire Instruction Constants of SimpleTrigger

SimpleTrigger Misfire Instructions

```
MISFIRE_INSTRUCTION_IGNORE_MISFIRE_POLICY
MISFIRE_INSTRUCTION_FIRE_NOW
MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_EXISTING_REPEAT_COUNT
MISFIRE_INSTRUCTION_RESCHEDULE_NOW_WITH_REMAINING_REPEAT_COUNT
MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_REMAINING_COUNT
MISFIRE_INSTRUCTION_RESCHEDULE_NEXT_WITH_EXISTING_COUNT
```

You should recall from the earlier lessons that all triggers have the *Trigger.MISFIRE_INSTRUCTION_SMART_POLICY* instruction available for use, and this instruction is also the default for all trigger types.

If the 'smart policy' instruction is used, SimpleTrigger dynamically chooses between its various MISFIRE instructions, based on the configuration and state of the given SimpleTrigger instance. The JavaDoc for the SimpleTrigger.updateAfterMisfire() method explains the exact details of this dynamic behavior.

When building SimpleTriggers, you specify the misfire instruction as part of the simple schedule (via SimpleSchedulerBuilder):

```
  trigger = newTrigger()
    .withIdentity("trigger7", "group1")
    .withSchedule(simpleSchedule()
        .withIntervalInMinutes(5)
        .repeatForever()
        .withMisfireHandlingInstructionNextWithExistingCount())
    .build();
```

Table of Contents | ‹ Lesson 5 | Lesson 7 ›

Tutorial

Lesson 6: CronTrigger

CronTrigger is often more useful than SimpleTrigger, if you need a job-firing schedule that recurs based on calendar-like notions, rather than on the exactly specified intervals of SimpleTrigger.

With CronTrigger, you can specify firing-schedules such as "every Friday at noon", or "every weekday and 9:30 am", or even "every 5 minutes between 9:00 am and 10:00 am on every Monday, Wednesday and Friday during January".

Even so, like SimpleTrigger, CronTrigger has a ***startTime*** which specifies when the schedule is in force, and an (optional) ***endTime*** that specifies when the schedule should be discontinued.

Cron Expressions

Cron-Expressions are used to configure instances of CronTrigger. Cron-Expressions are strings that are actually made up of seven sub-expressions, that describe individual details of the schedule. These sub-expression are separated with white-space, and represent:

1. Seconds
2. Minutes
3. Hours
4. Day-of-Month
5. Month
6. Day-of-Week
7. Year (optional field)

An example of a complete cron-expression is the string *"0 0 12 ? * WED"* - which means "every Wednesday at 12:00:00 pm".

Individual sub-expressions can contain ranges and/or lists. For example, the day of week field in the previous (which reads "WED") example could be replaced with "MON-FRI", "MON,WED,FRI", or even "MON-WED,SAT".

Wild-cards (the *'' character) can be used to say "every" possible value of this field. Therefore the "* character in the "Month" field of the previous example simply means "every month". A '*' in the Day-Of-Week field would therefore obviously mean "every day of the week".

All of the fields have a set of valid values that can be specified. These values should be fairly obvious - such as the numbers 0 to 59 for seconds and minutes, and the values 0 to 23 for hours. Day-of-Month can be any value 1-31, but you need to be careful about how many days are in a given month! Months can be specified as values between 0 and 11, or by using the strings JAN, FEB, MAR, APR, MAY, JUN, JUL, AUG, SEP, OCT, NOV and DEC. Days-of-Week can be specified as values between 1 and 7 (1 = Sunday) or by using the strings SUN, MON, TUE, WED, THU, FRI and SAT.

The '/' character can be used to specify increments to values. For example, if you put '0/15' in the Minutes field, it means 'every 15th minute of the hour, starting at minute zero'. If you used '3/20' in the Minutes field, it would mean 'every 20th minute of the hour, starting at minute three' - or in other words it is the same as

specifying '3,23,43' in the Minutes field. Note the subtlety that *"/35" does *not* mean "every 35 minutes" - it mean "every 35th minute of the hour, starting at minute zero" - or in other words the same as specifying '0,35'.

The '?' character is allowed for the day-of-month and day-of-week fields. It is used to specify "no specific value". This is useful when you need to specify something in one of the two fields, but not the other. See the examples below (and CronTrigger JavaDoc) for clarification.

The 'L' character is allowed for the day-of-month and day-of-week fields. This character is short-hand for "last", but it has different meaning in each of the two fields. For example, the value "L" in the day-of-month field means "the last day of the month" - day 31 for January, day 28 for February on non-leap years. If used in the day-of-week field by itself, it simply means "7" or "SAT". But if used in the day-of-week field after another value, it means "the last xxx day of the month" - for example "6L" or "FRIL" both mean "the last friday of the month". You can also specify an offset from the last day of the month, such as "L-3" which would mean the third-to-last day of the calendar month. *When using the 'L' option, it is important not to specify lists, or ranges of values, as you'll get confusing/unexpected results.*

The 'W' is used to specify the weekday (Monday-Friday) nearest the given day. As an example, if you were to specify "15W" as the value for the day-of-month field, the meaning is: "the nearest weekday to the 15th of the month".

The '#' is used to specify "the nth" XXX weekday of the month. For example, the value of "6#3" or "FRI#3" in the day-of-week field means "the third Friday of the month".

Here are a few more examples of expressions and their meanings - you can find even more in the JavaDoc for org.quartz.CronExpression

Example Cron Expressions

CronTrigger Example 1 - an expression to create a trigger that simply fires every 5 minutes

"0 0/5 * * * ?"

CronTrigger Example 2 - an expression to create a trigger that fires every 5 minutes, at 10 seconds after the minute (i.e. 10:00:10 am, 10:05:10 am, etc.).

"10 0/5 * * * ?"

CronTrigger Example 3 - an expression to create a trigger that fires at 10:30, 11:30, 12:30, and 13:30, on every Wednesday and Friday.

"0 30 10-13 ? * WED,FRI"

CronTrigger Example 4 - an expression to create a trigger that fires every half hour between the hours of 8 am and 10 am on the 5th and 20th of every month. Note that the trigger will NOT fire at 10:00 am, just at 8:00, 8:30, 9:00 and 9:30

"0 0/30 8-9 5,20 * ?"

Note that some scheduling requirements are too complicated to express with a single trigger - such as "every 5 minutes between 9:00 am and 10:00 am, and every 20 minutes between 1:00 pm and 10:00 pm". The solution

in this scenario is to simply create two triggers, and register both of them to run the same job.

Building CronTriggers

CronTrigger instances are built using TriggerBuilder (for the trigger's main properties) and CronScheduleBuilder (for the CronTrigger-specific properties). To use these builders in a DSL-style, use static imports:

```
import static org.quartz.TriggerBuilder.*;
import static org.quartz.CronScheduleBuilder.*;
import static org.quartz.DateBuilder.*:
```

Build a trigger that will fire every other minute, between 8am and 5pm, every day:

```
  trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .withSchedule(cronSchedule("0 0/2 8-17 * * ?"))
    .forJob("myJob", "group1")
    .build();
```

Build a trigger that will fire daily at 10:42 am:

```
  trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .withSchedule(dailyAtHourAndMinute(10, 42))
    .forJob(myJobKey)
    .build();
```

or -

```
  trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .withSchedule(cronSchedule("0 42 10 * * ?"))
    .forJob(myJobKey)
    .build();
```

Build a trigger that will fire on Wednesdays at 10:42 am, in a TimeZone other than the system's default:

```
  trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .withSchedule(weeklyOnDayAndHourAndMinute(DateBuilder.WEDNESDAY, 10, 42))
    .forJob(myJobKey)
    .inTimeZone(TimeZone.getTimeZone("America/Los_Angeles"))
    .build();
```

or -

```
  trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .withSchedule(cronSchedule("0 42 10 ? * WED"))
    .inTimeZone(TimeZone.getTimeZone("America/Los_Angeles"))
    .forJob(myJobKey)
    .build();
```

CronTrigger Misfire Instructions

The following instructions can be used to inform Quartz what it should do when a misfire occurs for CronTrigger. (Misfire situations were introduced in the More About Triggers section of this tutorial). These instructions are defined as constants on CronTrigger itself (including JavaDoc describing their behavior). The instructions include:

Misfire Instruction Constants of CronTrigger

```
MISFIRE_INSTRUCTION_IGNORE_MISFIRE_POLICY
MISFIRE_INSTRUCTION_DO_NOTHING
MISFIRE_INSTRUCTION_FIRE_NOW
```

All triggers also have the *Trigger.MISFIRE_INSTRUCTION_SMART_POLICY* instruction available for use, and this instruction is also the default for all trigger types. The 'smart policy' instruction is interpreted by CronTrigger as *MISFIRE_INSTRUCTION_FIRE_NOW*. The JavaDoc for the CronTrigger.updateAfterMisfire() method explains the exact details of this behavior.

When building CronTriggers, you specify the misfire instruction as part of the simple schedule (via CronSchedulerBuilder):

```
  trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .withSchedule(cronSchedule("0 0/2 8-17 * * ?")
        ..withMisfireHandlingInstructionFireAndProceed())
    .forJob("myJob", "group1")
    .build();
```

Table of Contents | ‹ Lesson 6 | Lesson 8 ›

Tutorial

Lesson 7: TriggerListeners and JobListeners

Listeners are objects that you create to perform actions based on events occurring within the scheduler. As you can probably guess, ***TriggerListeners*** receive events related to triggers, and ***JobListeners*** receive events related to jobs.

Trigger-related events include: trigger firings, trigger mis-firings (discussed in the "Triggers" section of this document), and trigger completions (the jobs fired off by the trigger is finished).

The org.quartz.TriggerListener Interface

```
public interface TriggerListener {

    public String getName();

    public void triggerFired(Trigger trigger, JobExecutionContext context);

    public boolean vetoJobExecution(Trigger trigger, JobExecutionContext context);

    public void triggerMisfired(Trigger trigger);

    public void triggerComplete(Trigger trigger, JobExecutionContext context,
            int triggerInstructionCode);
}
```

Job-related events include: a notification that the job is about to be executed, and a notification when the job has completed execution.

The org.quartz.JobListener Interface

```
public interface JobListener {

    public String getName();

    public void jobToBeExecuted(JobExecutionContext context);

    public void jobExecutionVetoed(JobExecutionContext context);

    public void jobWasExecuted(JobExecutionContext context,
            JobExecutionException jobException);

}
```

Using Your Own Listeners

To create a listener, simply create an object that implements the org.quartz.TriggerListener and/or org.quartz.JobListener interface. Listeners are then registered with the scheduler during run time, and must be given a name (or rather, they must advertise their own name via their getName() method).

For your convenience, tather than implementing those interfaces, your class could also extend the class JobListenerSupport or TriggerListenerSupport and simply override the events you're interested in.

Using Your Own Listeners

Listeners are registered with the scheduler's ListenerManager along with a Matcher that describes which Jobs/Triggers the listener wants to receive events for.

> Listeners are registered with the scheduler during run time, and are NOT stored in the JobStore along with the jobs and triggers. This is because listeners are typically an integration point with your application. Hence, each time your application runs, the listeners need to be re-registered with the scheduler.

Adding a JobListener that is interested in a particular job:

```
scheduler.getListenerManager().addJobListener(myJobListener, KeyMatcher.jobKeyEquals(new JobKey("
```

You may want to use static imports for the matcher and key classes, which will make your defining the matchers cleaner:

```
import static org.quartz.JobKey.*;
import static org.quartz.impl.matchers.KeyMatcher.*;
import static org.quartz.impl.matchers.GroupMatcher.*;
import static org.quartz.impl.matchers.AndMatcher.*;
import static org.quartz.impl.matchers.OrMatcher.*;
import static org.quartz.impl.matchers.EverythingMatcher.*;
...etc.
```

Which turns the above example into this:

```
scheduler.getListenerManager().addJobListener(myJobListener, jobKeyEquals(jobKey("myJobName", "my
```

Adding a JobListener that is interested in all jobs of a particular group:

```
scheduler.getListenerManager().addJobListener(myJobListener, jobGroupEquals("myJobGroup"));
```

Adding a JobListener that is interested in all jobs of two particular groups:

```
scheduler.getListenerManager().addJobListener(myJobListener, or(jobGroupEquals("myJobGroup"), job
```

Adding a JobListener that is interested in all jobs:

```
scheduler.getListenerManager().addJobListener(myJobListener, allJobs());
```

...Registering TriggerListeners works in just the same way.

Listeners are not used by most users of Quartz, but are handy when application requirements create the need for the notification of events, without the Job itself having to explicitly notify the application.

Table of Contents | ‹ Lesson 7 | Lesson 9 ›

Tutorial

Lesson 8: SchedulerListeners

SchedulerListeners are much like TriggerListeners and JobListeners, except they receive notification of events within the Scheduler itself - not necessarily events related to a specific trigger or job.

Scheduler-related events include: the addition of a job/trigger, the removal of a job/trigger, a serious error within the scheduler, notification of the scheduler being shutdown, and others.

The org.quartz.SchedulerListener Interface

```
public interface SchedulerListener {

    public void jobScheduled(Trigger trigger);

    public void jobUnscheduled(String triggerName, String triggerGroup);

    public void triggerFinalized(Trigger trigger);

    public void triggersPaused(String triggerName, String triggerGroup);

    public void triggersResumed(String triggerName, String triggerGroup);

    public void jobsPaused(String jobName, String jobGroup);

    public void jobsResumed(String jobName, String jobGroup);

    public void schedulerError(String msg, SchedulerException cause);

    public void schedulerStarted();

    public void schedulerInStandbyMode();

    public void schedulerShutdown();

    public void schedulingDataCleared();
}
```

SchedulerListeners are registered with the scheduler's ListenerManager. SchedulerListeners can be virtually any object that implements the org.quartz.SchedulerListener interface.

Adding a SchedulerListener:

```
scheduler.getListenerManager().addSchedulerListener(mySchedListener);
```

Removing a SchedulerListener:

```
scheduler.getListenerManager().removeSchedulerListener(mySchedListener);
```

Table of Contents | ‹ Lesson 8 | Lesson 10 ›

Tutorial

Lesson 9: Job Stores

JobStore's are responsible for keeping track of all the "work data" that you give to the scheduler: jobs, triggers, calendars, etc. Selecting the appropriate JobStore for your Quartz scheduler instance is an important step. Luckily, the choice should be a very easy one once you understand the differences between them. You declare which JobStore your scheduler should use (and it's configuration settings) in the properties file (or object) that you provide to the SchedulerFactory that you use to produce your scheduler instance.

> Never use a JobStore instance directly in your code. For some reason many people attempt to do this. The JobStore is for behind-the-scenes use of Quartz itself. You have to tell Quartz (through configuration) which JobStore to use, but then you should only work with the Scheduler interface in your code.

RAMJobStore

RAMJobStore is the simplest JobStore to use, it is also the most performant (in terms of CPU time). RAMJobStore gets its name in the obvious way: it keeps all of its data in RAM. This is why it's lightning-fast, and also why it's so simple to configure. The drawback is that when your application ends (or crashes) all of the scheduling information is lost - this means RAMJobStore cannot honor the setting of "non-volatility" on jobs and triggers. For some applications this is acceptable - or even the desired behavior, but for other applications, this may be disastrous.

To use RAMJobStore (and assuming you're using StdSchedulerFactory) simply specify the class name org.quartz.simpl.RAMJobStore as the JobStore class property that you use to configure quartz:

Configuring Quartz to use RAMJobStore

```
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

There are no other settings you need to worry about.

JDBCJobStore

JDBCJobStore is also aptly named - it keeps all of its data in a database via JDBC. Because of this it is a bit more complicated to configure than RAMJobStore, and it also is not as fast. However, the performance draw-back is not terribly bad, especially if you build the database tables with indexes on the primary keys. On fairly modern set of machines with a decent LAN (between the scheduler and database) the time to retrieve and update a firing trigger will typically be less than 10 milliseconds.

JDBCJobStore works with nearly any database, it has been used widely with Oracle, PostgreSQL, MySQL, MS SQLServer, HSQLDB, and DB2. To use JDBCJobStore, you must first create a set of database tables for Quartz to use. You can find table-creation SQL scripts in the "docs/dbTables" directory of the Quartz distribution. If there is not already a script for your database type, just look at one of the existing ones, and modify it in any way necessary for your DB. One thing to note is that in these scripts, all the the tables start with the prefix "QRTZ_" (such as the tables "QRTZ_TRIGGERS", and "QRTZ_JOB_DETAIL"). This prefix can actually be anything you'd like, as long as you inform JDBCJobStore what the prefix is (in your Quartz properties). Using different prefixes may be useful for creating multiple sets of tables, for multiple scheduler

instances, within the same database.

Once you've got the tables created, you have one more major decision to make before configuring and firing up JDBCJobStore. You need to decide what type of transactions your application needs. If you don't need to tie your scheduling commands (such as adding and removing triggers) to other transactions, then you can let Quartz manage the transaction by using ***JobStoreTX*** as your JobStore (this is the most common selection).

If you need Quartz to work along with other transactions (i.e. within a J2EE application server), then you should use ***JobStoreCMT*** - in which case Quartz will let the app server container manage the transactions.

The last piece of the puzzle is setting up a DataSource from which JDBCJobStore can get connections to your database. DataSources are defined in your Quartz properties using one of a few different approaches. One approach is to have Quartz create and manage the DataSource itself - by providing all of the connection information for the database. Another approach is to have Quartz use a DataSource that is managed by an application server that Quartz is running inside of - by providing JDBCJobStore the JNDI name of the DataSource. For details on the properties, consult the example config files in the "docs/config" folder.

To use JDBCJobStore (and assuming you're using StdSchedulerFactory) you first need to set the JobStore class property of your Quartz configuration to be either org.quartz.impl.jdbcjobstore.JobStoreTX or org.quartz.impl.jdbcjobstore.JobStoreCMT - depending on the selection you made based on the explanations in the above few paragraphs.

Configuring Quartz to use JobStoreTx

```
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
```

Next, you need to select a DriverDelegate for the JobStore to use. The DriverDelegate is responsible for doing any JDBC work that may be needed for your specific database. StdJDBCDelegate is a delegate that uses "vanilla" JDBC code (and SQL statements) to do its work. If there isn't another delegate made specifically for your database, try using this delegate - we've only made database-specific delegates for databases that we've found problems using StdJDBCDelegate with (which seems to be most!). Other delegates can be found in the "org.quartz.impl.jdbcjobstore" package, or in its sub-packages. Other delegates include DB2v6Delegate (for DB2 version 6 and earlier), HSQLDBDelegate (for HSQLDB), MSSQLDelegate (for Microsoft SQLServer), PostgreSQLDelegate (for PostgreSQL), WeblogicDelegate (for using JDBC drivers made by Weblogic), OracleDelegate (for using Oracle), and others.

Once you've selected your delegate, set its class name as the delegate for JDBCJobStore to use.

Configuring JDBCJobStore to use a DriverDelegate

```
org.quartz.jobStore.driverDelegateClass = org.quartz.impl.jdbcjobstore.StdJDBCDelegate
```

Next, you need to inform the JobStore what table prefix (discussed above) you are using.

Configuring JDBCJobStore with the Table Prefix

```
org.quartz.jobStore.tablePrefix = QRTZ_
```

And finally, you need to set which DataSource should be used by the JobStore. The named DataSource must also be defined in your Quartz properties. In this case, we're specifying that Quartz should use the DataSource name "myDS" (that is defined elsewhere in the configuration properties).

Configuring JDBCJobStore with the name of the DataSource to use

```
org.quartz.jobStore.dataSource = myDS
```

If your Scheduler is busy (i.e. nearly always executing the same number of jobs as the size of the thread pool, then you should probably set the number of connections in the DataSource to be the about the size of the thread pool + 2.

The "org.quartz.jobStore.useProperties" config parameter can be set to "true" (defaults to false) in order to instruct JDBCJobStore that all values in JobDataMaps will be Strings, and therefore can be stored as name-value pairs, rather than storing more complex objects in their serialized form in the BLOB column. This is much safer in the long term, as you avoid the class versioning issues that there are with serializing your non-String classes into a BLOB.

TerracottaJobStore

TerracottaJobStore is new with Quartz 1.7. It provides a means for scaling and robustness without the use of a database. This means your database can be kept free of load from Quartz, and can instead have all of its resources saved for the rest of your application.

TerracottaJobStore can be ran clustered or non-clustered, and in either case provides a storage medium for your job data that is persistent between application restarts, because the data is stored in the Terracotta server. It's performance is much better than using a database via JDBCJobStore (about an order of magnitude better), but fairly slower than RAMJobStore.

To use TerracottaJobStore (and assuming you're using StdSchedulerFactory) simply specify the class name org.quartz.jobStore.class = org.terracotta.quartz.TerracottaJobStore as the JobStore class property that you use to configure quartz, and add one extra line of configuration to specify the location of the Terracotta server:

Configuring Quartz to use TerracottaJobStore

```
org.quartz.jobStore.class = org.terracotta.quartz.TerracottaJobStore
org.quartz.jobStore.tcConfigUrl = localhost:9510
```

More information about this JobStore and Terracotta can be found at http://www.terracotta.org/quartz

Table of Contents | ‹ Lesson 9 | Lesson 11 ›

Tutorial

Lesson 10: Configuration, Resource Usage and SchedulerFactory

The architecture of Quartz is modular, and therefore to get it running several components need to be "snapped" together. Fortunately, some helpers exist for making this happen.

The major components that need to be configured before Quartz can do its work are:

- ThreadPool
- JobStore
- DataSources (if necessary)
- The Scheduler itself

The ***ThreadPool*** provides a set of Threads for Quartz to use when executing Jobs. The more threads in the pool, the greater number of Jobs that can run concurrently. However, too many threads may bog-down your system. Most Quartz users find that 5 or so threads are plenty- because they have fewer than 100 jobs at any given time, the jobs are not generally scheduled to run at the same time, and the jobs are short-lived (complete quickly). Other users find that they need 10, 15, 50 or even 100 threads - because they have tens-of-thousands of triggers with various schedules - which end up having an average of between 10 and 100 jobs trying to execute at any given moment. Finding the right size for your scheduler's pool is completely dependent on what you're using the scheduler for. There are no real rules, other than to keep the number of threads as small as possible (for the sake of your machine's resources) - but make sure you have enough for your Jobs to fire on time. Note that if a trigger's time to fire arrives, and there isn't an available thread, Quartz will block (pause) until a thread comes available, then the Job will execute - some number of milliseconds later than it should have. This may even cause the thread to misfire - if there is no available thread for the duration of the scheduler's configured "misfire threshold".

A ThreadPool interface is defined in the org.quartz.spi package, and you can create a ThreadPool implementation in any way you like. Quartz ships with a simple (but very satisfactory) thread pool named org.quartz.simpl.SimpleThreadPool. This ThreadPool simply maintains a fixed set of threads in its pool - never grows, never shrinks. But it is otherwise quite robust and is very well tested - as nearly everyone using Quartz uses this pool.

JobStores and ***DataSources*** were discussed in Lesson 9 of this tutorial. Worth noting here, is the fact that all JobStores implement the org.quartz.spi.JobStore interface - and that if one of the bundled JobStores does not fit your needs, then you can make your own.

Finally, you need to create your ***Scheduler*** instance. The Scheduler itself needs to be given a name, told its RMI settings, and handed instances of a JobStore and ThreadPool. The RMI settings include whether the Scheduler should create itself as an RMI server object (make itself available to remote connections), what host and port to use, etc.. StdSchedulerFactory (discussed below) can also produce Scheduler instances that are actually proxies (RMI stubs) to Schedulers created in remote processes.

StdSchedulerFactory

StdSchedulerFactory is an implementation of the org.quartz.SchedulerFactory interface. It uses a set of properties (java.util.Properties) to create and initialize a Quartz Scheduler. The properties are generally stored

in and loaded from a file, but can also be created by your program and handed directly to the factory. Simply calling getScheduler() on the factory will produce the scheduler, initialize it (and its ThreadPool, JobStore and DataSources), and return a handle to its public interface.

There are some sample configurations (including descriptions of the properties) in the "docs/config" directory of the Quartz distribution. You can find complete documentation in the "Configuration" manual under the "Reference" section of the Quartz documentation.

DirectSchedulerFactory

DirectSchedulerFactory is another SchedulerFactory implementation. It is useful to those wishing to create their Scheduler instance in a more programmatic way. Its use is generally discouraged for the following reasons: (1) it requires the user to have a greater understanding of what they're doing, and (2) it does not allow for declarative configuration - or in other words, you end up hard-coding all of the scheduler's settings.

Logging

Quartz uses the SLF4J framework for all of its logging needs. In order to "tune" the logging settings (such as the amount of output, and where the output goes), you need to understand the SLF4J framework, which is beyond the scope of this document.

If you want to capture extra information about trigger firings and job executions, you may be interested in enabling the *org.quartz.plugins.history.LoggingJobHistoryPlugin* and/or *org.quartz.plugins.history.LoggingTriggerHistoryPlugin*.

Table of Contents | ‹ Lesson 10 | Lesson 12 ›

Tutorial

Lesson 11: Advanced (Enterprise) Features

Clustering

Clustering currently works with the JDBC-Jobstore (JobStoreTX or JobStoreCMT) and the TerracottaJobStore. Features include load-balancing and job fail-over (if the JobDetail's "request recovery" flag is set to true).

Clustering With JobStoreTX or JobStoreCMT

Enable clustering by setting the "org.quartz.jobStore.isClustered" property to "true". Each instance in the cluster should use the same copy of the quartz.properties file. Exceptions of this would be to use properties files that are identical, with the following allowable exceptions: Different thread pool size, and different value for the "org.quartz.scheduler.instanceId" property. Each node in the cluster MUST have a unique instanceId, which is easily done (without needing different properties files) by placing "AUTO" as the value of this property.

> Never run clustering on separate machines, unless their clocks are synchronized using some form of time-sync service (daemon) that runs very regularly (the clocks must be within a second of each other). See http://www.boulder.nist.gov/timefreq/service/its.htm if you are unfamiliar with how to do this.
>
> Never fire-up a non-clustered instance against the same set of tables that any other instance is running against. You may get serious data corruption, and will definitely experience erratic behavior.

Only one node will fire the job for each firing. What I mean by that is, if the job has a repeating trigger that tells it to fire every 10 seconds, then at 12:00:00 exactly one node will run the job, and at 12:00:10 exactly one node will run the job, etc. It won't necessarily be the same node each time - it will more or less be random which node runs it. The load balancing mechanism is near-random for busy schedulers (lots of triggers) but favors the same node that just was just active for non-busy (e.g. one or two triggers) schedulers.

Clustering With TerracottaJobStore

Simply configure the scheduler to use TerracottaJobStore (covered in Lesson 9: JobStores), and your scheduler will be all set for clustering.

You may also want to consider implications of how you setup your Terracotta server, particularly configuration options that turn on features such as storing data on disk, utilization of fsync, and running an array of Terracotta servers for HA.

More information about this JobStore and Terracotta can be found at http://www.terracotta.org/quartz

JTA Transactions

As explained in Lesson 9: JobStores, JobStoreCMT allows Quartz scheduling operations to be performed within larger JTA transactions.

Jobs can also execute within a JTA transaction (UserTransaction) by setting the "org.quartz.scheduler.wrapJobExecutionInUserTransaction" property to "true". With this option set, a a JTA transaction will begin() just before the Job's execute method is called, and commit() just after the call to execute terminates.

Aside from Quartz automatically wrapping Job executions in JTA transactions, calls you make on the Scheduler interface also participate in transactions when using JobStoreCMT. Just make sure you've started a transaction before calling a method on the scheduler. You can do this either directly, through the use of UserTransaction, or by putting your code that uses the scheduler within a SessionBean that uses container managed transactions.

Table of Contents | ‹ Lesson 11

Tutorial

Lesson 12: Miscellaneous Features of Quartz

Plug-Ins

Quartz provides an interface (org.quartz.spi.SchedulerPlugin) for plugging-in additional functionality.

Plugins that ship with Quartz to provide various utility capabilities can be found documented in the ***org.quartz.plugins*** package. They provide functionality such as auto-scheduling of jobs upon scheduler startup, logging a history of job and trigger events, and ensuring that the scheduler shuts down cleanly when the JVM exits.

JobFactory

When a trigger fires, the Job it is associated to is instantiated via the JobFactory configured on the Scheduler. The default JobFactory simply calls newInstance() on the job class. You may want to create your own implementation of JobFactory to accomplish things such as having your application's IoC or DI container produce/initialize the job instance.

See the **org.quartz.spi.JobFactory** interface, and the associated **Scheduler.setJobFactory(fact)** method.

'Factory-Shipped' Jobs

Quartz also provides a number of utility Jobs that you can use in your application for doing things like sending e-mails and invoking EJBs. These out-of-the-box Jobs can be found documented in the ***org.quartz.jobs*** package.

CronTrigger Tutorial

Introduction

`cron` is a UNIX tool that has been around for a long time, so its scheduling capabilities are powerful and proven. The `CronTrigger` class is based on the scheduling capabilities of cron.

`CronTrigger` uses "cron expressions", which are able to create firing schedules such as: "At 8:00am every Monday through Friday" or "At 1:30am every last Friday of the month".

Cron expressions are powerful, but can be pretty confusing. This tutorial aims to take some of the mystery out of creating a cron expression, giving users a resource which they can visit before having to ask in a forum or mailing list.

Format

A cron expression is a string comprised of 6 or 7 fields separated by white space. Fields can contain any of the allowed values, along with various combinations of the allowed special characters for that field. The fields are as follows:

Field Name	Mandatory	Allowed Values	Allowed Special Characters
Seconds	YES	0-59	, - * /
Minutes	YES	0-59	, - * /
Hours	YES	0-23	, - * /
Day of month	YES	1-31	, - * ? / L W
Month	YES	1-12 or JAN-DEC	, - * /
Day of week	YES	1-7 or SUN-SAT	, - * ? / L #
Year	NO	empty, 1970-2099	, - * /

So cron expressions can be as simple as this: `* * * * ? *`

or more complex, like this: `0/5 14,18,3-39,52 * ? JAN,MAR,SEP MON-FRI 2002-2010`

Special characters

- **`*`** (*"all values"*) - used to select all values within a field. For example, "*" in the minute field means *"every minute"*.
- **`?`** (*"no specific value"*) - useful when you need to specify something in one of the two fields in which the character is allowed, but not the other. For example, if I want my trigger to fire on a particular day of the month (say, the 10th), but don't care what day of the week that happens to be, I would put "10" in the day-of-month field, and "?" in the day-of-week field. See the examples below for clarification.
- **`-`** - used to specify ranges. For example, "10-12" in the hour field means *"the hours 10, 11 and 12"*.
- **`,`** - used to specify additional values. For example, "MON,WED,FRI" in the day-of-week field means *"the days Monday, Wednesday, and Friday"*.
- **`/`** - used to specify increments. For example, "0/15" in the seconds field means *"the seconds 0, 15, 30,*

and 45". And "5/15" in the seconds field means *"the seconds 5, 20, 35, and 50"*. You can also specify '/' after the **" character - in this case "** is equivalent to having '0' before the '/'. '1/3' in the day-of-month field means *"fire every 3 days starting on the first day of the month"*.

- **`L`** (*"last"*) - has different meaning in each of the two fields in which it is allowed. For example, the value "L" in the day-of-month field means *"the last day of the month"* - day 31 for January, day 28 for February on non-leap years. If used in the day-of-week field by itself, it simply means "7" or "SAT". But if used in the day-of-week field after another value, it means *"the last xxx day of the month"* - for example "6L" means *"the last friday of the month"*. You can also specify an offset from the last day of the month, such as "L-3" which would mean the third-to-last day of the calendar month. *When using the 'L' option, it is important not to specify lists, or ranges of values, as you'll get confusing/unexpected results.*
- **`W`** (*"weekday"*) - used to specify the weekday (Monday-Friday) nearest the given day. As an example, if you were to specify "15W" as the value for the day-of-month field, the meaning is: *"the nearest weekday to the 15th of the month"*. So if the 15th is a Saturday, the trigger will fire on Friday the 14th. If the 15th is a Sunday, the trigger will fire on Monday the 16th. If the 15th is a Tuesday, then it will fire on Tuesday the 15th. However if you specify "1W" as the value for day-of-month, and the 1st is a Saturday, the trigger will fire on Monday the 3rd, as it will not 'jump' over the boundary of a month's days. The 'W' character can only be specified when the day-of-month is a single day, not a range or list of days.

  The 'L' and 'W' characters can also be combined in the day-of-month field to yield 'LW', which translates to *"last weekday of the month"*.

- **`#`** - used to specify "the nth" XXX day of the month. For example, the value of "6#3" in the day-of-week field means *"the third Friday of the month"* (day 6 = Friday and "#3" = the 3rd one in the month). Other examples: "2#1" = the first Monday of the month and "4#5" = the fifth Wednesday of the month. Note that if you specify "#5" and there is not 5 of the given day-of-week in the month, then no firing will occur that month.

  The legal characters and the names of months and days of the week are not case sensitive. `MON` is the same as `mon`.

Examples

Here are some full examples:

Expression	**Meaning**
`0 0 12 * * ?`	Fire at 12pm (noon) every day
`0 15 10 ? * *`	Fire at 10:15am every day
`0 15 10 * * ?`	Fire at 10:15am every day
`0 15 10 * * ? *`	Fire at 10:15am every day
`0 15 10 * * ? 2005`	Fire at 10:15am every day during the year 2005
`0 * 14 * * ?`	Fire every minute starting at 2pm and ending at 2:59pm, every day
`0 0/5 14 * * ?`	Fire every 5 minutes starting at 2pm and ending at 2:55pm, every day
`0 0/5 14,18 * * ?`	Fire every 5 minutes starting at 2pm and ending at 2:55pm, AND fire every 5 minutes starting at 6pm and ending at 6:55pm, every day

Examples

`0 0-5 14 * * ?`	Fire every minute starting at 2pm and ending at 2:05pm, every day
`0 10,44 14 ? 3 WED`	Fire at 2:10pm and at 2:44pm every Wednesday in the month of March.
`0 15 10 ? * MON-FRI`	Fire at 10:15am every Monday, Tuesday, Wednesday, Thursday and Friday
`0 15 10 15 * ?`	Fire at 10:15am on the 15th day of every month
`0 15 10 L * ?`	Fire at 10:15am on the last day of every month
`0 15 10 L-2 * ?`	Fire at 10:15am on the 2nd-to-last last day of every month
`0 15 10 ? * 6L`	Fire at 10:15am on the last Friday of every month
`0 15 10 ? * 6L`	Fire at 10:15am on the last Friday of every month
`0 15 10 ? * 6L 2002-2005`	Fire at 10:15am on every last friday of every month during the years 2002, 2003, 2004 and 2005
`0 15 10 ? * 6#3`	Fire at 10:15am on the third Friday of every month
`0 0 12 1/5 * ?`	Fire at 12pm (noon) every 5 days every month, starting on the first day of the month.
`0 11 11 11 11 ?`	Fire every November 11th at 11:11am.

Pay attention to the effects of '?' and '*' in the day-of-week and day-of-month fields!

Notes

- Support for specifying both a day-of-week and a day-of-month value is not complete (you must currently use the '?' character in one of these fields).
- Be careful when setting fire times between the hours of the morning when "daylight savings" changes occur in your locale (for US locales, this would typically be the hour before and after 2:00 AM - because the time shift can cause a skip or a repeat depending on whether the time moves back or jumps forward. You may find this wikipedia entry helpful in determining the specifics to your locale: https://secure.wikimedia.org/wikipedia/en/wiki/Daylight_saving_time_around_the_world

Examples Overview

Welcome to the documentation for the Quartz Example programs. As of version 1.5, Quartz ships with 13 out-of-the-box examples that demonstrate the various features of Quartz and the Quartz API.

Where to Find the Examples

All of the examples listed below are part of the Quartz distribution.

To download Quartz, visit http://www.quartz-scheduler.org/download and select the latest Quartz distribution.

The quartz examples are listed under the **examples** directory under the main Quartz directory. Under the **examples** directory, you will find an example sub-directory for each example, labeled **example1**, **example2**, **example3** etc...

Every example contains UNIX/Linux shell scripts for executing the examples as well at Windows batch files. Additionally, every example has a readme.txt file. Please consult this file before running the examples.

The source code for the examples are located in package **org.quartz.examples**. Every example has its own sub-package, **org.quartz.examples.example1**, **org.quartz.examples.example2**, etc...

Here we give an overview of each example program:

The Examples

Title	Description
Example 1 - First Quartz Program	Think of this as a "Hello World" for Quartz
Example 2 - Simple Triggers	Shows a dozen different ways of using Simple Triggers to schedule your jobs
Example 3 - Cron Triggers	Shows how Cron Triggers can be used to schedule your job
Example 4 - Job State and Parameters	Demonstrates how parameters can be passed into jobs and how jobs maintain state
Example 5 - Handling Job Misfires	Sometimes job will not execute when they are supposed to. See how to handle these Misfires
Example 6 - Dealing with Job Exceptions	No job is perfect. See how you can let the scheduler know how to deal with exceptions that are thrown by your job
Exampl 7 - Interrupting Jobs	Shows how the scheduler can interrupt your jobs and how to code your jobs to deal with interruptions
Example 8 - Fun with Calendars	Demonstrates how a Holiday calendar can be used to exclude execution of jobs on a holiday
Example 9 - Job Listeners	Use job listeners to have one job trigger another job, building a simple workflow
Example 10 - Using Quartz Plug-Ins	Demonstrates the use of the XML Job Initialization plug-in as well as the History Logging plug-ins
Example 11 - Quartz Under High Load	Quartz can run a lot of jobs but see how thread pools can limit how many jobs can execute simultaneously

Example 12 - Remote Job Scheduling using RMI	Using Remote Method Invocation, a Quartz scheduler can be remotely scheduled by a client
Example 13 - Clustered Quartz	Demonstrates how Quartz can be used in a clustered environment and how Quartz can use the database to persist scheduling information
Example 14 - Trigger Priorities	Demonstrates how Trigger priorities can be used to manage firing order for Triggers with the same fire time
Example 15 - TC Clustered Quartz	Demonstrates how Quartz can be clustered with Terracotta, rather than with a database

Contents | Next ›

Example - Your First Quartz Program

This example is designed to demonstrate how to get up and running with Quartz. This example will fire off a simple job that says "Hello World".

The program will perform the following actions:

- Start up the Quartz Scheduler
- Schedule a job to run at the next even minute
- Wait for 90 seconds to give Quartz a chance to run the job
- Shut down the Scheduler

Running the Example

This example can be executed from the **examples/example1** directory. There are two out-of-the-box methods for running this example

- **example1.sh** - A UNIX/Linux shell script
- **example1.bat** - A Windows Batch file

The Code

The code for this example resides in the package **org.quartz.examples.example1**.

The code in this example is made up of the following classes:

Class Name	Description
SimpleExample	The main program
HelloJob	A simple job that says Hello World

HelloJob

HelloJob is a simple job that implements the *Job* interface and logs a nice message to the log (by default, this will simply go to the screen). The current date and time is printed in the job so that you can see exactly when the job is run.

```
public void execute(JobExecutionContext context) throws JobExecutionException {
    // Say Hello to the World and display the date/time
    _log.info("Hello World! - " + new Date());
```

```
}
```

SimpleExample

The program starts by getting an instance of the Scheduler. This is done by creating a *StdSchedulerFactory* and then using it to create a scheduler. This will create a simple, RAM-based scheduler.

```
SchedulerFactory sf = new StdSchedulerFactory();
Scheduler sched = sf.getScheduler();
```

The HelloJob is defined as a Job to Quartz using the *JobDetail* class:

```
// define the job and tie it to our HelloJob class
JobDetail job = newJob(HelloJob.class)
    .withIdentity("job1", "group1")
    .build();
```

We create a *SimpleTrigger* that will fire off at the next round minute:

```
// compute a time that is on the next round minute
Date runTime = evenMinuteDate(new Date());

// Trigger the job to run on the next round minute
Trigger trigger = newTrigger()
    .withIdentity("trigger1", "group1")
    .startAt(runTime)
    .build();
```

We now will associate the Job to the Trigger in the scheduler:

```
// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger);
```

At this point, the job has been schedule to run when its trigger fires. However, the scheduler is not yet running. So, we must tell the scheduler to start up!

```
sched.start();
```

To let the program have an opportunity to run the job, we then sleep for 90 seconds. The scheduler is running in the background and should fire off the job during those 90 seconds.

```
Thread.sleep(90L * 1000L);
```

Finally, we will gracefully shutdown the scheduler:

```
sched.shutdown(true);
```

Note: passing *true* into the *shutdown* message tells the Quartz Scheduler to wait until all jobs have completed running before returning from the method call.

Contents | ‹ Prev | Next ›

Example - Cron-based Triggers

This example is designed to demonstrate how you can use Cron Triggers to schedule jobs. This example will fire off several simple jobs that say "Hello World" and display the date and time that the job was executed.

The program will perform the following actions:

- Start up the Quartz Scheduler
- Schedule several jobs using various features of CronTrigger
- Wait for 300 seconds (5 minutes) to give Quartz a chance to run the jobs
- Shut down the Scheduler

Note: Refer to the Quartz javadoc for a thorough explanation of CronTrigger.

Running the Example

This example can be executed from the **examples/example3** directory. There are two out-of-the-box methods for running this example

- **example3.sh** - A UNIX/Linux shell script
- **example3.bat** - A Windows Batch file

The Code

The code for this example resides in the package **org.quartz.examples.example3**.

The code in this example is made up of the following classes:

Class Name	Description
CronTriggerExample	The main program
SimpleJob	A simple job that says Hello World and displays the date/time

SimpleJob

SimpleJob is a simple job that implements the *Job* interface and logs a nice message to the log (by default, this will simply go to the screen). The current date and time is printed in the job so that you can see exactly when the job is run.

```
public void execute(JobExecutionContext context) throws JobExecutionException {
    JobKey jobKey = context.getJobDetail().getKey();
    _log.info("SimpleJob says: " + jobKey + " executing at " + new Date());
}
```

CronTriggerExample

The program starts by getting an instance of the Scheduler. This is done by creating a *StdSchedulerFactory* and then using it to create a scheduler. This will create a simple, RAM-based scheduler.

```
SchedulerFactory sf = new StdSchedulerFactory();
Scheduler sched = sf.getScheduler();
```

Job #1 is scheduled to run every 20 seconds

```
JobDetail job = newJob(SimpleJob.class)
    .withIdentity("job1", "group1")
    .build();

CronTrigger trigger = newTrigger()
    .withIdentity("trigger1", "group1")
    .withSchedule(cronSchedule("0/20 * * * * ?"))
    .build();

sched.scheduleJob(job, trigger);
```

Job #2 is scheduled to run every other minute, starting at 15 seconds past the minute.

```
job = newJob(SimpleJob.class)
    .withIdentity("job2", "group1")
    .build();

trigger = newTrigger()
    .withIdentity("trigger2", "group1")
    .withSchedule(cronSchedule("15 0/2 * * * ?"))
    .build();

sched.scheduleJob(job, trigger);
```

Job #3 is scheduled to every other minute, between 8am and 5pm (17 o'clock).

```
job = newJob(SimpleJob.class)
    .withIdentity("job3", "group1")
    .build();

trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .withSchedule(cronSchedule("0 0/2 8-17 * * ?"))
    .build();

sched.scheduleJob(job, trigger);
```

Job #4 is scheduled to run every three minutes but only between 5pm and 11pm

```
job = newJob(SimpleJob.class)
    .withIdentity("job4", "group1")
    .build();

trigger = newTrigger()
    .withIdentity("trigger4", "group1")
    .withSchedule(cronSchedule("0 0/3 17-23 * * ?"))
    .build();

sched.scheduleJob(job, trigger);
```

Job #5 is scheduled to run at 10am on the 1st and 15th days of the month

```
job = newJob(SimpleJob.class)
    .withIdentity("job5", "group1")
    .build();

trigger = newTrigger()
```

```
    .withIdentity("trigger5", "group1")
    .withSchedule(cronSchedule("0 0 10am 1,15 * ?"))
    .build();

sched.scheduleJob(job, trigger);
```

Job #6 is scheduled to run every 30 seconds on Weekdays (Monday through Friday)

```
job = newJob(SimpleJob.class)
    .withIdentity("job6", "group1")
    .build();

trigger = newTrigger()
    .withIdentity("trigger6", "group1")
    .withSchedule(cronSchedule("0,30 * * ? * MON-FRI"))
    .build();

sched.scheduleJob(job, trigger);
```

Job #7 is scheduled to run every 30 seconds on Weekends (Saturday and Sunday)

```
job = newJob(SimpleJob.class)
    .withIdentity("job7", "group1")
    .build();

trigger = newTrigger()
    .withIdentity("trigger7", "group1")
    .withSchedule(cronSchedule("0,30 * * ? * SAT,SUN"))
    .build();

sched.scheduleJob(job, trigger);
```

The scheduler is then started (it also would have been fine to start it before scheduling the jobs).

```
sched.start();
```

To let the program have an opportunity to run the job, we then sleep for five minutes (300 seconds). The scheduler is running in the background and should fire off several jobs during that time.

Note: Because many of the jobs have hourly and daily restrictions on them, not all of the jobs will run in this example. For example: Job #6 only runs on Weekdays while Job #7 only runs on Weekends.

```
Thread.sleep(300L * 1000L);
```

Finally, we will gracefully shutdown the scheduler:

```
sched.shutdown(true);
```

Note: passing *true* into the *shutdown* message tells the Quartz Scheduler to wait until all jobs have completed running before returning from the method call.

Contents | ‹ Prev | Next ›

Example - Job Parameters and Job State

This example is designed to demonstrate how you can pass run-time parameters into quartz jobs and how you can maintain state in a job.

The program will perform the following actions:

- Start up the Quartz Scheduler
- Schedule two jobs, each job will execute the every ten seconds for a total of times
- The scheduler will pass a run-time job parameter of "Green" to the first job instance
- The scheduler will pass a run-time job parameter of "Red" to the second job instance
- The program will wait 60 seconds so that the two jobs have plenty of time to run
- Shut down the Scheduler

Running the Example

This example can be executed from the **examples/example4** directory. There are two out-of-the-box methods for running this example

- **example4.sh** - A UNIX/Linux shell script
- **example4.bat** - A Windows Batch file

The Code

The code for this example resides in the package **org.quartz.examples.example4**.

The code in this example is made up of the following classes:

Class Name	Description
JobStateExample	The main program
ColorJob	A simple job that prints a favorite color (passed in as a run-time parameter) and displays its execution count.

ColorJob

ColorJob is a simple class that implement the Job interface, and is annotated as such:

```
@PersistJobDataAfterExecution
@DisallowConcurrentExecution
public class ColorJob implements Job {
```

The annotations cause behavior just as their names describe - multiple instances of the job will not be allowed to run concurrently (consider a case where a job has code in its execute() method that takes 34 seconds to run, but it is scheduled with a trigger that repeats every 30 seconds), and will have its JobDataMap contents re-persisted in the scheduler's JobStore after each execution. For the purposes of this example, only *@PersistJobDataAfterExecution* annotation is truly relevant, but it's always wise to use the *@DisallowConcurrentExecution* annotation with it, to prevent race-conditions on saved data.

ColorJob logs the following information when the job is executed:

- The job's identification key (name and group) and time/date of execution
- The job's favorite color (which is passed in as a run-time parameter)
- The job's execution count calculated from a member variable
- The job's execution count maintained as a job map parameter

```
_log.info("ColorJob: " + jobKey + " executing at " + new Date() + "\n" +
    "  favorite color is " + favoriteColor + "\n" +
    "  execution count (from job map) is " + count + "\n" +
    "  execution count (from job member variable) is " + _counter);
```

The variable *favoriteColor* is passed in as a job parameter. It is retrieved as follows from the *JobDataMap*:

```
JobDataMap data = context.getJobDetail().getJobDataMap();
String favoriteColor = data.getString(FAVORITE_COLOR);
```

The variable *count* is stored in the job data map as well:

```
JobDataMap data = context.getJobDetail().getJobDataMap();
int count = data.getInt(EXECUTION_COUNT);
```

The variable is later incremented and stored back into the job data map so that job state can be preserved:

```
count++;
data.put(EXECUTION_COUNT, count);
```

There is also a member variable named *counter*. This variable is defined as a member variable to the class:

```
private int _counter = 1;
```

This variable is also incremented and displayed. However, its count will always be displayed as "1" because Quartz will always instantiate a new instance of the class during each execution - which prevents member variables from being used to maintain state.

JobStateExample

The program starts by getting an instance of the Scheduler. This is done by creating a *StdSchedulerFactory* and then using it to create a scheduler. This will create a simple, RAM-based scheduler.

```
SchedulerFactory sf = new StdSchedulerFactory();
Scheduler sched = sf.getScheduler();
```

Job #1 is scheduled to run every 10 seconds, for a total of five times:

```
JobDetail job1 = newJob(ColorJob.class)
    .withIdentity("job1", "group1")
    .build();

SimpleTrigger trigger1 = newTrigger()
    .withIdentity("trigger1", "group1")
    .startAt(startTime)
    .withSchedule(simpleSchedule()
            .withIntervalInSeconds(10)
            .withRepeatCount(4))
    .build();
```

Job #1 is passed in two job parameters. One is a favorite color, with a value of "Green". The other is an execution count, which is initialized with a value of 1.

```
job1.getJobDataMap().put(ColorJob.FAVORITE_COLOR, "Green");
job1.getJobDataMap().put(ColorJob.EXECUTION_COUNT, 1);
```

Job #2 is also scheduled to run every 10 seconds, for a total of five times:

```
JobDetail job2 = newJob(ColorJob.class)
    .withIdentity("job2", "group1")
    .build();

SimpleTrigger trigger2 = newTrigger()
    .withIdentity("trigger2", "group1")
    .startAt(startTime)
    .withSchedule(simpleSchedule()
            .withIntervalInSeconds(10)
            .withRepeatCount(4))
    .build();
```

Job #2 is also passed in two job parameters. One is a favorite color, with a value of "Red". The other is an execution count, which is initialized with a value of 1.

```
job2.getJobDataMap().put(ColorJob.FAVORITE_COLOR, "Red");
job2.getJobDataMap().put(ColorJob.EXECUTION_COUNT, 1);
```

The scheduler is then started.

```
sched.start();
```

To let the program have an opportunity to run the job, we then sleep for one minute (60 seconds)

```
Thread.sleep(60L * 1000L);
```

Finally, we will gracefully shutdown the scheduler:

```
sched.shutdown(true);
```

Note: passing *true* into the *shutdown* message tells the Quartz Scheduler to wait until all jobs have completed running before returning from the method call.

Contents | ‹ Prev | Next ›

Example - Job Misfires

This example is designed to demonstrate concepts related to trigger misfires.

The program will perform the following actions:

- Start up the Quartz Scheduler
- Schedule two jobs, each job will execute the every three seconds, indefinitely
- The jobs will take ten seconds to run (preventing the execution trigger from firing every three seconds)
- Each job has different misfire instructions

- The program will wait 10 minutes so that the two jobs have plenty of time to run
- Shut down the Scheduler

Running the Example

This example can be executed from the **examples/example5** directory. There are two out-of-the-box methods for running this example

- **example5.sh** - A UNIX/Linux shell script
- **example5.bat** - A Windows Batch file

The Code

The code for this example resides in the package **org.quartz.examples.example5**.

The code in this example is made up of the following classes:

Class Name	Description
MisfireExample	The main program
StatefulDumbJob	A simple job class who's execute method takes 10 seconds to run

StatefulDumbJob

StatefulDumbJob is a simple job that prints its execution time and then will wait for a period of time before completing.
The amount of wait time is defined by the job parameter EXECUTION_DELAY. If this job parameter is not passed in, the job will default to a wait time of 5 seconds. The job is also keep its own count of how many times it has executed using a value in its JobDataMap called NUM_EXECUTIONS. Because the class has the *PersistJobDataAfterExecution* annotation, the execution count is preserved between each execution.

```
@PersistJobDataAfterExecution
@DisallowConcurrentExecution
public class StatefulDumbJob implements Job {

    public static final String NUM_EXECUTIONS = "NumExecutions";
    public static final String EXECUTION_DELAY = "ExecutionDelay";

    public StatefulDumbJob() {
    }
    public void execute(JobExecutionContext context)
        throws JobExecutionException {
        System.err.println("---" + context.getJobDetail().getKey()
                + " executing.[" + new Date() + "]");

        JobDataMap map = context.getJobDetail().getJobDataMap();

        int executeCount = 0;
        if (map.containsKey(NUM_EXECUTIONS)) {
            executeCount = map.getInt(NUM_EXECUTIONS);
        }

        executeCount++;
        map.put(NUM_EXECUTIONS, executeCount);
```

```
        long delay = 50001;
        if (map.containsKey(EXECUTION_DELAY)) {
            delay = map.getLong(EXECUTION_DELAY);
        }

        try {
            Thread.sleep(delay);
        } catch (Exception ignore) {
        }

        System.err.println("  -" + context.getJobDetail().getKey()
                + " complete (" + executeCount + ").");
    }
}
```

MisfireExample

The program starts by getting an instance of the Scheduler. This is done by creating a *StdSchedulerFactory* and then using it to create a scheduler. This will create a simple, RAM-based scheduler because no specific quartz.properties config file telling it to do otherwise is provided.

```
SchedulerFactory sf = new StdSchedulerFactory();
Scheduler sched = sf.getScheduler();
```

Job #1 is scheduled to run every 3 seconds indefinitely. An execution delay of 10 seconds is passed into the job:

```
JobDetail job = newJob(StatefulDumbJob.class)
    .withIdentity("statefulJob1", "group1")
    .usingJobData(StatefulDumbJob.EXECUTION_DELAY, 10000L)
    .build();

SimpleTrigger trigger = newTrigger()
    .withIdentity("trigger1", "group1")
    .startAt(startTime)
    .withSchedule(simpleSchedule()
            .withIntervalInSeconds(3)
            .repeatForever())
    .build();

sched.scheduleJob(job, trigger);
```

Job #2 is scheduled to run every 3 seconds indefinitely. An execution delay of 10 seconds is passed into the job:

```
job = newJob(StatefulDumbJob.class)
            .withIdentity("statefulJob2", "group1")
            .usingJobData(StatefulDumbJob.EXECUTION_DELAY, 10000L)
            .build();

        trigger = newTrigger()
            .withIdentity("trigger2", "group1")
            .startAt(startTime)
            .withSchedule(simpleSchedule()
                    .withIntervalInSeconds(3)
                    .repeatForever()
                    .withMisfireHandlingInstructionNowWithExistingCount()) // set misfire instruc
            .build();
```

Note: The trigger for job #2 is set with a misfire instruction that will cause it to reschedule with the existing repeat count. This policy forces quartz to refire the trigger as soon as possible. Job #1 uses the default "smart" misfire policy for simple triggers, which causes the trigger to fire at it's next normal execution time.

The scheduler is then started.

```
sched.start();
```

To let the program have an opportunity to run the job, we then sleep for ten minutes (600 seconds)

```
Thread.sleep(600L * 1000L);
```

Finally, we will gracefully shutdown the scheduler:

```
sched.shutdown(true);
```

Note: passing *true* into the *shutdown* message tells the Quartz Scheduler to wait until all jobs have completed running before returning from the method call.

Contents | ‹ Prev | Next ›

Example - Dealing with Job Exceptions

This example is designed to demonstrate how can deal with job execution exceptions. Jobs in Quartz are permitted to throw a *JobExecutionExceptions*. When this exception is thrown, you can instruct quartz what action to take.

The program will perform the following actions:

- Start up the Quartz Scheduler
- Schedule two jobs, each job will execute the every three seconds, indefintely
- The jobs will throw an exception, and quartz will take appropriate action
- The program will wait 60 seconds so that the two jobs have plenty of time to run
- Shut down the Scheduler

Running the Example

This example can be executed from the **examples/example6** directory. There are two out-of-the-box methods for running this example

- **example6.sh** - A UNIX/Linux shell script
- **example6.bat** - A Windows Batch file

The Code

The code for this example resides in the package **org.quartz.examples.example6**.

The code in this example is made up of the following classes:

Class Name	Description

JobExceptionExample	The main program
BadJob1	A simple job that will throw an exception and instruct quartz to refire its trigger immediately
BadJob2	A simple job that will throw an exception and instruct quartz to never schedule the job again

BadJob1

BadJob1 is a simple job that simply creates an artificial exception (divide by zero). When this exception is caught, a *JobExecutionException* is thrown and set to refire the job immediatly.

```
try {
    int zero = 0;
    int calculation = 4815 / zero;
}
catch (Exception e) {
    _log.info("--- Error in job!");
    JobExecutionException e2 =
        new JobExecutionException(e);
    // this job will refire immediately
    e2.refireImmediately();
    throw e2;
}
```

This will force quartz to run this job over and over and over and over again.

BadJob2

BadJob2 is a simple job that simply creates an artificial exception (divide by zero). When this exception is caught, a *JobExecutionException* is thrown and set to ensure that quartz never runs the job again.

```
try {
    int zero = 0;
    int calculation = 4815 / zero;
}
catch (Exception e) {
    _log.info("--- Error in job!");
    JobExecutionException e2 =
        new JobExecutionException(e);
    // Quartz will automatically unschedule
    // all triggers associated with this job
    // so that it does not run again
    e2.setUnscheduleAllTriggers(true);
    throw e2;
}
```

This will force quartz to shutdown this job so that it does not run again.

JobExceptionExample

The program starts by getting an instance of the Scheduler. This is done by creating a *StdSchedulerFactory* and then using it to create a scheduler. This will create a simple, RAM-based scheduler.

```
SchedulerFactory sf = new StdSchedulerFactory();
Scheduler sched = sf.getScheduler();
```

Job #1 is scheduled to run every 3 seconds indefinitely. This job will fire *BadJob1*.

```
JobDetail job = newJob(BadJob1.class)
    .withIdentity("badJob1", "group1")
    .build();

SimpleTrigger trigger = newTrigger()
    .withIdentity("trigger1", "group1")
    .startAt(startTime)
    .withSchedule(simpleSchedule()
            .withIntervalInSeconds(3)
            .repeatForever())
    .build();

Date ft = sched.scheduleJob(job, trigger);
```

Job #2 is scheduled to run every 3 seconds indefinitely. This job will fire *BadJob2*.

```
job = newJob(BadJob2.class)
    .withIdentity("badJob2", "group1")
    .build();

trigger = newTrigger()
    .withIdentity("trigger2", "group1")
    .startAt(startTime)
    .withSchedule(simpleSchedule()
            .withIntervalInSeconds(3)
            .repeatForever())
    .build();

ft = sched.scheduleJob(job, trigger);
```

The scheduler is then started.

```
sched.start();
```

To let the program have an opportunity to run the job, we then sleep for 1 minute (60 seconds)

```
Thread.sleep(60L * 1000L);
```

This scheduler will run both jobs (BadJob1 and BadJob2). Both jobs will throw an exception. Job 1 should attempt to refire immediately. Job 2 should never run again.

Finally, we will gracefully shutdown the scheduler:

```
sched.shutdown(true);
```

Note: passing *true* into the *shutdown* message tells the Quartz Scheduler to wait until all jobs have completed running before returning from the method call.

Contents | ‹ Prev

Example - Trigger Priorities

This example will demonstrate how Trigger priorities can be used to manage firing order for Triggers with the same fire time.

The program will perform the following actions:

- Create a Scheduler with a single worker thread
- Schedule three Triggers with different priorities that fire the first time at the same time, and a second time at staggered intervals
- Start up the Quartz Scheduler
- Wait for 30 seconds to give Quartz a chance to fire the Triggers
- Shut down the Scheduler

Running the Example

This example can be executed from the **examples/example14** directory. There are two out-of-the-box methods for running this example

- **example14.sh** - A UNIX/Linux shell script
- **example14.bat** - A Windows Batch file

Expected Results

Each of the three Triggers should fire twice. Once in order of priority as they all start at the same time, and a second time in order of their staggered firing times. You should see something like this in the log or on the console:

```
INFO 15 Aug 12:15:51.345 PM PriorityExampleScheduler_Worker-0 org.quartz.examples.example14.Trigg
TRIGGER: Priority10Trigger15SecondRepeat
INFO 15 Aug 12:15:51.345 PM PriorityExampleScheduler_Worker-0 org.quartz.examples.example14.Trigg
TRIGGER: Priority5Trigger10SecondRepeat
INFO 15 Aug 12:15:51.345 PM PriorityExampleScheduler_Worker-0 org.quartz.examples.example14.Trigg
TRIGGER: PriorityNeg5Trigger5SecondRepeat
INFO 15 Aug 12:15:56.220 PM PriorityExampleScheduler_Worker-0 org.quartz.examples.example14.Trigg
TRIGGER: PriorityNeg5Trigger5SecondRepeat
INFO 15 Aug 12:16:01.220 PM PriorityExampleScheduler_Worker-0 org.quartz.examples.example14.Trigg
TRIGGER: Priority5Trigger10SecondRepeat
INFO 15 Aug 12:16:06.220 PM PriorityExampleScheduler_Worker-0 org.quartz.examples.example14.Trigg
TRIGGER: Priority10Trigger15SecondRepeat
```

The Code

The code for this example resides in the package **org.quartz.examples.example14**.

The code in this example is made up of the following classes:

Class Name	Description
PriorityExample	The main program
TriggerEchoJob	A simple job that echos the name if the Trigger that fired it

TriggerEchoJob

TriggerEchoJob is a simple job that implements the *Job* interface and logs the name of the *Trigger* that fired it to the log (by default, this will simply go to the screen):

```
public void execute(JobExecutionContext context) throws JobExecutionException {
```

```
    LOG.info("TRIGGER: " + context.getTrigger().getKey());
}
```

PriorityExample

The program starts by getting an instance of the Scheduler. This is done by creating a *StdSchedulerFactory* and then using it to create a scheduler.

```
SchedulerFactory sf = new StdSchedulerFactory(
    "org/quartz/examples/example14/quartz_priority.properties");
Scheduler sched = sf.getScheduler();
```

We pass a specific Quartz properties file to the StdSchedulerFactory to configure our new Scheduler instance. These properties will create a simple, RAM-based scheduler with only one worker thread so that we can see priorities act as the tie breaker when Triggers compete for the single thread, *quartz_priority.properties*:

```
org.quartz.scheduler.instanceName=PriorityExampleScheduler
# Set thread count to 1 to force Triggers scheduled for the same time to
# to be ordered by priority.
org.quartz.threadPool.threadCount=1
org.quartz.threadPool.class=org.quartz.simpl.SimpleThreadPool

org.quartz.jobStore.class=org.quartz.simpl.RAMJobStore
```

The TriggerEchoJob is defined as a Job to Quartz using the *JobDetail* class. It passes **null** for its group, so it will use the default group:JobDetail job = new JobDetail("TriggerEchoJob", null, TriggerEchoJob.class);

We create three *SimpleTrigger*s that will all fire the first time five seconds from now but with different priorities, and then fire a second time at staggered five second intervals:

```
// Calculate the start time of all triggers as 5 seconds from now
Date startTime = futureDate(5, IntervalUnit.SECOND);

// First trigger has priority of 1, and will repeat after 5 seconds
Trigger trigger1 = newTrigger()
    .withIdentity("PriorityNeg5Trigger5SecondRepeat")
    .startAt(startTime)
    .withSchedule(simpleSchedule().withRepeatCount(1).withIntervalInSeconds(5))
    .withPriority(1)
    .forJob(job)
    .build();

// Second trigger has default priority of 5 (default), and will repeat after 10 seconds
Trigger trigger2 = newTrigger()
    .withIdentity("Priority5Trigger10SecondRepeat")
    .startAt(startTime)
    .withSchedule(simpleSchedule().withRepeatCount(1).withIntervalInSeconds(10))
    .forJob(job)
    .build();

// Third trigger has priority 10, and will repeat after 15 seconds
Trigger trigger3 = newTrigger()
    .withIdentity("Priority10Trigger15SecondRepeat")
    .startAt(startTime)
    .withSchedule(simpleSchedule().withRepeatCount(1).withIntervalInSeconds(15))
    .withPriority(10)
    .forJob(job)
    .build();
```

We now associate the three Triggers with our Job in the scheduler. The first time we need to also add the job itself to the scheduler:

```
// Tell quartz to schedule the job using our trigger
sched.scheduleJob(job, trigger1);
sched.scheduleJob(trigger2);
sched.scheduleJob(trigger3);
```

At this point, the triggers have been scheduled to run. However, the scheduler is not yet running. So, we must tell the scheduler to start up!

```
sched.start();
```

To let the program have an opportunity to run the job, we then sleep for 30 seconds. The scheduler is running in the background and should fire off the job six times during those 30 seconds.

```
Thread.sleep(30L * 1000L);
```

Finally, we will gracefully shutdown the scheduler:

```
sched.shutdown(true);
```

Note: passing *true* into the *shutdown* message tells the Quartz Scheduler to wait until all jobs have completed running before returning from the method call.

Quartz Enterprise Job Scheduler Cookbook

The Quartz cookbook is a collection of succinct code examples of doing specific things with Quartz.

The examples assume you have used static imports of Quartz's DSL classes such as these:

```
import static org.quartz.JobBuilder.*;
import static org.quartz.TriggerBuilder.*;
import static org.quartz.SimpleScheduleBuilder.*;
import static org.quartz.CronScheduleBuilder.*;
import static org.quartz.CalendarIntervalScheduleBuilder.*;
import static org.quartz.JobKey.*;
import static org.quartz.TriggerKey.*;
import static org.quartz.DateBuilder.*;
import static org.quartz.impl.matchers.KeyMatcher.*;
import static org.quartz.impl.matchers.GroupMatcher.*;
import static org.quartz.impl.matchers.AndMatcher.*;
import static org.quartz.impl.matchers.OrMatcher.*;
import static org.quartz.impl.matchers.EverythingMatcher.*;
```

Choose from the following menu of How-Tos:

Contents | Next ›

How-To: Instantiating a Scheduler

Instantiating the Default Scheduler

```
// the 'default' scheduler is defined in "quartz.properties" found
// in the current working directory, in the classpath, or
// resorts to a fall-back default that is in the quartz.jar

SchedulerFactory sf = new StdSchedulerFactory();
Scheduler scheduler = sf.getScheduler();

// Scheduler will not execute jobs until it has been started (though they can be scheduled before
scheduler.start();
```

Instantiating A Specific Scheduler From Specific Properties

```
StdSchedulerFactory sf = new StdSchedulerFactory();

sf.initialize(schedulerProperties);

Scheduler scheduler = sf.getScheduler();

// Scheduler will not execute jobs until it has been started (though they can be scheduled before
scheduler.start();
```

Instantiating A Specific Scheduler From A Specific Property File

```
StdSchedulerFactory sf = new StdSchedulerFactory();

sf.initialize(fileName);

Scheduler scheduler = sf.getScheduler();

// Scheduler will not execute jobs until it has been started (though they can be scheduled before
scheduler.start();
```

Contents | ‹ Prev | Next ›

How-To: Using Multiple (Non-Clustered) Schedulers

Reasons you may want to do this:

- For managing resources - e.g. if you have a mix of light-weight and heavy-weight jobs, then you may wish to have a scheduler with many threads to service the lightweight jobs and one with few threads to service the heavy-weight jobs, in order to keep your machines resources from being overwhelmed by running to many heavy-weight jobs concurrently.
- To schedule jobs in one application, but have them execute within another (when using JDBC-JobStore).

Note that you can create as many schedulers as you like within any application, but they must have unique scheduler names (typically defined in the quartz.properties file). This means that you'll need to have multiple properties files, which means that you'll need to specify them as you initialize the StdSchedulerFactory (as it only defaults to finding "quartz.properties").

If you run multiple schedulers they can of course all have distinct characteristics - e.g. one may use RAMJobStore and have 100 worker threads, and another may use JDBC-JobStore and have 20 worker threads.

> Never start (scheduler.start()) a non-clustered instance against the same set of database tables that any other instance with the same scheduler name is running (start()ed) against. You may get serious data corruption, and will definitely experience erratic behavior.

Example/Discussion Relating To Scheduling Jobs From One Application To Be Executed In Another Application

This description/usage applies to JDBC-JobStore. You may also want to look at RMI or JMX features to control a Scheduler in a remote process - which works for any JobStore. You may also be interested in the Terracotta Quartz Where features.

Currently, If you want to have particular jobs run in a particular scheduler, then it needs to be a distinct scheduler - unless you use the Terracotta Quartz Where features.

Suppose you have an application "App A" that needs to schedule jobs (based on user input) that need to run either on the local process/machine "Machine A" (for simple jobs) or on a remote machine "Machine B" (for complex jobs).

It is possible within an application to instantiate two (or more) schedulers, and schedule jobs into both (or more) schedulers, and have only the jobs placed into one scheduler run on the local machine. This is achieved by calling scheduler.start() on the scheduler(s) within the process where you want the jobs to execute. Scheduler.start() causes the scheduler instance to start processing the jobs (i.e. start waiting for trigger fire times to arrive, and then executing the jobs). However a non-started scheduler instance can still be used to schedule (and retrieve) jobs.

For example:

- In "App A" create "Scheduler A" (with config that points it at database tables prefixed with "A"), and invoke start() on "Scheduler A". Now "Scheduler A" in "App A" will execute jobs scheduled by "Scheduler A" in "App A"
- In "App A" create "Scheduler B" (with config that points it at database tables prefixed with "B"), and DO NOT invoke start() on "Scheduler B". Now "Scheduler B" in "App A" can schedule jobs to be ran where "Scheduler B" is started.
- In "App B" create "Scheduler B" (with config that points it at database tables prefixed with "B"), and invoke start() on "Scheduler B". Now "Scheduler B" in "App B" will execute jobs scheduled by "Scheduler B" in "App A".

Contents | ‹ Prev | Next ›

How-To: Using Scheduler Listeners

Creating a SchedulerListener

Extend TriggerListenerSupport and override methods for events you're interested in.

```
package foo;

import org.quartz.Trigger;
import org.quartz.listeners.SchedulerListenerSupport;

public class MyOtherSchedulerListener extends SchedulerListenerSupport {

    @Override
    public void schedulerStarted() {
        // do something with the event
    }

    @Override
    public void schedulerShutdown() {
        // do something with the event
    }

    @Override
    public void jobScheduled(Trigger trigger) {
        // do something with the event
    }

}
```

Registering A SchedulerListener With The Scheduler

```
scheduler.getListenerManager().addSchedulerListener(mySchedListener);
```

Contents | ‹ Prev | Next ›

How-To: Placing a Scheduler in Stand-by Mode

Placing a Scheduler in Stand-by Mode

```
// start() was previously invoked on the scheduler

scheduler.standby();

// now the scheduler will not fire triggers / execute jobs

// ...

scheduler.start();

// now the scheduler will fire triggers and execute jobs
```

Contents | ‹ Prev | Next ›

How-To: Initializing a scheduler within a servlet container

There are two approaches for this which are shown below.

For both cases, make sure to look at the JavaDOC for the related classes to see all possible configuration parameters, as a complete set is not show below.

Adding A Context/Container Listener To web.xml

```
...
     <context-param>
         <param-name>quartz:config-file</param-name>
         <param-value>/some/path/my_quartz.properties</param-value>
     </context-param>
     <context-param>
         <param-name>quartz:shutdown-on-unload</param-name>
         <param-value>true</param-value>
     </context-param>
     <context-param>
         <param-name>quartz:wait-on-shutdown</param-name>
         <param-value>false</param-value>
     </context-param>
     <context-param>
         <param-name>quartz:start-scheduler-on-load</param-name>
         <param-value>true</param-value>
     </context-param>
...
     <listener>
         <listener-class>
             org.quartz.ee.servlet.QuartzInitializerListener
         </listener-class>
     </listener>
...
```

Adding A Start-up Servlet To web.xml

```
...
    <servlet>
      <servlet-name>QuartzInitializer</servlet-name>
      <servlet-class>org.quartz.ee.servlet.QuartzInitializerServlet</servlet-class>
      <init-param>

        <param-name>shutdown-on-unload</param-name>
        <param-value>true</param-value>
      </init-param>
      <load-on-startup>2</load-on-startup>

    </servlet>
...
```

How-To: Shutting Down a Scheduler

To shutdown / destroy a scheduler, simply call one of the shutdown(..) methods.

Once you have shutdown a scheduler, it cannot be restarted (as threads and other resources are permanently destroyed). Also see the suspend method if you wish to simply pause the scheduler for a while.

Wait for Executing Jobs to Finish

```
//shutdown() does not return until executing Jobs complete execution
scheduler.shutdown(true);
```

Do Not Wait for Executing Jobs to Finish

```
//shutdown() returns immediately, but executing Jobs continue running to completion
scheduler.shutdown();
//or
scheduler.shutdown(false);
```

If you are using the `org.quartz.ee.servlet.QuartzInitializerListener` to fire up a scheduler in your servlet container, its `contextDestroyed()` method will shutdown the scheduler when your application is undeployed or the application server shuts down (unless its shutdown-on-unload property has been explicitly set to false).

Contents | ‹ Prev | Next ›

How-To: Defining a Job (with input data)

A Job Class

```
public class PrintPropsJob implements Job {

    public PrintPropsJob() {
        // Instances of Job must have a public no-argument constructor.
    }

    public void execute(JobExecutionContext context)
            throws JobExecutionException {

        JobDataMap data = context.getMergedJobDataMap();
        System.out.println("someProp = " + data.getString("someProp"));
    }

}
```

Defining a Job Instance

```
// Define job instance
JobDetail job1 = newJob(MyJobClass.class)
    .withIdentity("job1", "group1")
    .usingJobData("someProp", "someValue")
    .build();
```

Also note that if your Job class contains setter methods that match your JobDataMap keys (e.g. "setSomeProp" for the data in the above example), and you use the default JobFactory implementation, then Quartz will automatically call the setter method with the JobDataMap value, and there is no need to have code in the Job's execute method that retrieves the value from the JobDataMap.

Contents | ‹ Prev | Next ›

How-To: Scheduling a Job

Scheduling a Job

```
// Define job instance
JobDetail job1 = newJob(ColorJob.class)
    .withIdentity("job1", "group1")
    .build();

// Define a Trigger that will fire "now", and not repeat
Trigger trigger = newTrigger()
    .withIdentity("trigger1", "group1")
    .startNow()
    .build();

// Schedule the job with the trigger
sched.scheduleJob(job, trigger);
```

Contents | ‹ Prev | Next ›

How-To: Storing a Job for Later Use

Storing a Job

```
// Define a durable job instance (durable jobs can exist without triggers)
JobDetail job1 = newJob(MyJobClass.class)
    .withIdentity("job1", "group1")
    .storeDurably()
    .build();

// Add the the job to the scheduler's store
sched.addJob(job, false);
```

Contents | ‹ Prev | Next ›

How-To: Scheduling an already stored job

Scheduling an already stored job

```
// Define a Trigger that will fire "now" and associate it with the existing job
Trigger trigger = newTrigger()
    .withIdentity("trigger1", "group1")
    .startNow()
    .forJob(jobKey("job1", "group1"))
    .build();

// Schedule the trigger
sched.scheduleJob(trigger);
```

Contents | ‹ Prev | Next ›

How-To: Unscheduling a Job

Unscheduling a Particular Trigger of Job

```
// Unschedule a particular trigger from the job (a job may have more than one trigger)
scheduler.unscheduleJob(triggerKey("trigger1", "group1"));
```

Deleting a Job and Unscheduling All of Its Triggers

```
// Schedule the job with the trigger
scheduler.deleteJob(jobKey("job1", "group1"));
```

Contents | ‹ Prev | Next ›

How-To: Initializing Job Data With Scheduler Initialization

You can initialize the scheduler with predefined jobs and triggers using the XMLSchedulingDataProcessorPlugin (which, with the 1.8 release, replaced the older JobInitializationPlugin). An example is provided in the Quartz distribution in the directory examples/example10. However, following is a short description of how the plugin works.

First of all, we need to explicitly specify in the scheduler properties that we want to use the XMLSchedulingDataProcessorPlugin. This is an excerpt from an example quartz.properties:

```
#============================================================
# Configure the Job Initialization Plugin
#============================================================

org.quartz.plugin.jobInitializer.class = org.quartz.plugins.xml.XMLSchedulingDataProcessorPlugin
org.quartz.plugin.jobInitializer.fileNames = jobs.xml
org.quartz.plugin.jobInitializer.failOnFileNotFound = true
org.quartz.plugin.jobInitializer.scanInterval = 10
org.quartz.plugin.jobInitializer.wrapInUserTransaction = false
```

Let's see what each property does:

- **fileNames**: a comma separated list of filenames (with paths). These files contain the xml definition of jobs and associated triggers. We'll see an example jobs.xml definition shortly.
- **failOnFileNotFound**: if the xml definition files are not found, should the plugin throw an exception, thus preventing itself (the plugin) from initializing?
- **scanInterval**: the xml definition files can be reloaded if a file change is detected. This is the interval (in seconds) the files are looked at. Set to 0 to disable scanning.
- **wrapInUserTransaction**: if using the XMLSchedulingDataProcessorPlugin with JobStoreCMT, be sure to set the value of this property to true, otherwise you might experience unexpected behavior.

The jobs.xml file (or any other name you use for it in the fileNames property) declaratively defines jobs and triggers. It can also contain directive to delete existing data. Here's a self-explanatory example:

```
<?xml version='1.0' encoding='utf-8'?>
<job-scheduling-data xmlns="http://www.quartz-scheduler.org/xml/JobSchedulingData"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://www.quartz-scheduler.org/xml/JobSchedulingData http://www.quartz-sch
  version="1.8">

    <schedule>
        <job>
            <name>my-very-clever-job</name>
            <group>MYJOB_GROUP</group>

            <description>The job description</description>
            <job-class>com.acme.scheduler.job.CleverJob</job-class>
            <job-data-map allows-transient-data="false">

                <entry>
                    <key>burger-type</key>
                    <value>hotdog</value>
                </entry>
```

```
                <entry>

                    <key>dressing-list</key>
                    <value>ketchup,mayo</value>
                </entry>
            </job-data-map>
        </job>

        <trigger>
            <cron>
                <name>my-trigger</name>
                <group>MYTRIGGER_GROUP</group>
                <job-name>my-very-clever-job</job-name>

                <job-group>MYJOB_GROUP</job-group>
                <!-- trigger every night at 4:30 am -->
                <!-- do not forget to light the kitchen's light -->
                <cron-expression>0 30 4 * * ?</cron-expression>

            </cron>
        </trigger>
    </schedule>
</job-scheduling-data>
```

A further jobs.xml example is in the examples/example10 directory of the Quartz distribution.

Checkout the XML schema for full details of what is possible.

Contents | ‹ Prev | Next ›

How-To: Using Job Listeners

Creating a JobListener

Implement the JobListener interface.

```
package foo;

import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.quartz.JobListener;

public class MyJobListener implements JobListener {

    private String name;

    public MyJobListener(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void jobToBeExecuted(JobExecutionContext context) {
        // do something with the event
    }

    public void jobWasExecuted(JobExecutionContext context,
            JobExecutionException jobException) {
        // do something with the event
    }

    public void jobExecutionVetoed(JobExecutionContext context) {
        // do something with the event
    }
}
```

OR -

Extend JobListenerSupport.

```
package foo;

import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;
import org.quartz.listeners.JobListenerSupport;

public class MyOtherJobListener extends JobListenerSupport {

    private String name;

    public MyOtherJobListener(String name) {
        this.name = name;
    }
```

```
    public String getName() {
        return name;
    }
        @Override
    public void jobWasExecuted(JobExecutionContext context,
            JobExecutionException jobException) {
        // do something with the event
    }
}
```

Registering A JobListener With The Scheduler To Listen To All Jobs

```
scheduler.getListenerManager().addJobListener(myJobListener, allJobs());
```

Registering A JobListener With The Scheduler To Listen To A Specific Job

```
scheduler.getListenerManager().addJobListener(myJobListener, jobKeyEquals(jobKey("myJobName", "my
```

Registering A JobListener With The Scheduler To Listen To All Jobs In a Group

```
scheduler.getListenerManager().addJobListener(myJobListener, jobGroupEquals("myJobGroup"));
```

Contents | ‹ Prev | Next ›

How-To: Finding Triggers of a Job

Finding Triggers of a Job

```
List<Trigger> jobTriggers = sched.getTriggersOfJob(jobKey("jobName", "jobGroup"));
```

Contents | ‹ Prev | Next ›

How-To: Listing Jobs in the Scheduler

Listing all Jobs in the scheduler

```
// enumerate each job group
for(String group: sched.getJobGroupNames()) {
    // enumerate each job in group
    for(JobKey jobKey : sched.getJobKeys(groupEquals(group))) {
        System.out.println("Found job identified by: " + jobKey);
    }
}
```

Contents | ‹ Prev | Next ›

How-To: Update an existing job

Update an existing job

```
// Add the new job to the scheduler, instructing it to "replace"
//  the existing job with the given name and group (if any)
JobDetail job1 = newJob(MyJobClass.class)
    .withIdentity("job1", "group1")
    .build();

// store, and set overwrite flag to 'true'
scheduler.addJob(job1, true);
```

Contents | ‹ Prev | Next ›

How-To: Trigger That Executes Every 2 Days

At first glance, you may be tempted to use a CronTrigger. However, if this is truly to be every two days, CronTrigger won't work. To illustrate this, simply think of how many days are in a typical month (28-31). A cron expression like "0 0 5 2/2 * ?" would give us a trigger that would restart its count at the beginning of every month. This means that we would would get subsequent firings on July 30 and August 2, which is an interval of three days, not two.

Likewise, an expression like "0 0 5 1/2 * ?" would end up firing on July 31 and August 1, just one day apart.

Therefore, for this schedule, using SimpleTrigger or CalendarIntervalTrigger makes sense:

Using SimpleTrigger

Create a SimpleTrigger that executes 3:00PM tomorrow, and then every 48 hours (which may not always be at 3:00 PM - because adding 24 hours on days where daylight savings time shifts may result in 2:00 PM or 4:00 PM depending upon whether the 3:00 PM time was started during DST or standard time):

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // first fire time 15:00:00 tomorrow
    .withSchedule(simpleSchedule()
            .withIntervalInHours(2 * 24) // interval is actually set at 48 hours' worth of millis
            .repeatForever())
    .build();
```

Using CalendarIntervalTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // 15:00:00 tomorrow
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInDays(2)) // interval is set in calendar days
    .build();
```

Contents | ‹ Prev | Next ›

How-To: Trigger That Executes Every 2 Weeks

As with a trigger meant to fire every two days, CronTrigger won't work for this schedule. For more details, see Trigger That Fires Every 2 Days. We'll need to use a SimpleTrigger or CalendarIntervalTrigger:

Using SimpleTrigger

Create a SimpleTrigger that executes 3:00PM tomorrow, and then every 48 hours (which may not always be at 3:00 PM - because adding 24 hours on days where daylight savings time shifts may result in 2:00 PM or 4:00 PM depending upon whether the 3:00 PM time was started during DST or standard time):

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // first fire time 15:00:00 tomorrow
    .withSchedule(simpleSchedule()
            .withIntervalInHours(14 * 24) // interval is actually set at 14 * 24 hours' worth of
            .repeatForever())
    .build();
```

Using CalendarIntervalTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // 15:00:00 tomorrow
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInWeeks(2)) // interval is set in calendar weeks
    .build();
```

Contents | ‹ Prev | Next ›

How-To: Trigger That Executes Every Day

If you want a trigger that always fires at a certain time of day, use CronTrigger or CalendarIntervalTrigger because they can preserve the fire time's time of day across daylight savings time changes.

Using CronTrigger

Create a CronTrigger. that executes every day at 3:00PM:

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(dailyAtHourAndMinute(15, 0)) // fire every day at 15:00
    .build();
```

Using SimpleTrigger

Create a SimpleTrigger that executes 3:00PM tomorrow, and then every 24 hours (which may not always be at 3:00 PM - because adding 24 hours on days where daylight savings time shifts may result in 2:00 PM or 4:00 PM depending upon whether the 3:00 PM time was started during DST or standard time):

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // first fire time 15:00:00 tomorrow
    .withSchedule(simpleSchedule()
            .withIntervalInHours(24) // interval is actually set at 24 hours' worth of millisecon
            .repeatForever())
    .build();
```

Using CalendarIntervalTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // 15:00:00 tomorrow
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInDays(1)) // interval is set in calendar days
    .build();
```

Contents | ‹ Prev | Next ›

How-To: Listing Triggers In Scheduler

Listing all Triggers in the scheduler

```
// enumerate each trigger group
for(String group: sched.getTriggerGroupNames()) {
    // enumerate each trigger in group
    for(TriggerKey triggerKey : sched.getTriggerKeys(groupEquals(group))) {
        System.out.println("Found trigger identified by: " + triggerKey);
    }
}
```

Contents | ‹ Prev

How-To: Trigger That Executes Every Month

Using CronTrigger

Create a CronTrigger. that executes every Wednesday at 3:00PM:

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(monthlyOnDayAndHourAndMinute(5, 15, 0)) // fire on the 5th day of every month a
    .build();
```

OR -

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(cronSchedule("0 0 15 5 * ?")) // fire on the 5th day of every month at 15:00
    .build();
```

OR -

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(cronSchedule("0 0 15 L * ?")) // fire on the last day of every month at 15:00
    .build();
```

OR -

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(cronSchedule("0 0 15 LW * ?")) // fire on the last weekday day of every month a
    .build();
```

OR -

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(cronSchedule("0 0 15 L-3 * ?")) // fire on the third to last day of every month
    .build();
```

There are other possible combinations as well, which are more fully covered in the API documentation. All of these options were made by simply changing the day-of-month field. Imagine what you can do if you leverage

the other fields as well!

Using CalendarIntervalTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // 15:00:00 tomorrow
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInMonths(1)) // interval is set in calendar months
    .build();
```

Contents | ‹ Prev | Next ›

How-To: Trigger That Executes Every 90 minutes

Using SimpleTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(simpleSchedule()
            .withIntervalInMinutes(90)
            .repeatForever())
    .build();
```

Using CalendarIntervalTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInMinutes(90))
    .build();
```

Contents | ‹ Prev | Next ›

How-To: Trigger That Executes Every Ten Seconds

Using SimpleTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(simpleSchedule()
            .withIntervalInSeconds(10)
            .repeatForever())
    .build();
```

Contents | ‹ Prev | Next ›

How-To: Using Trigger Listeners

Creating a TriggerListener

Implement the TriggerListener interface.

```
package foo;

import org.quartz.JobExecutionContext;
import org.quartz.Trigger;
import org.quartz.TriggerListener;
import org.quartz.Trigger.CompletedExecutionInstruction;

public class MyTriggerListener implements TriggerListener {

    private String name;

    public MyTriggerListener(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    public void triggerComplete(Trigger trigger, JobExecutionContext context,
            CompletedExecutionInstruction triggerInstructionCode) {
        // do something with the event

    }

    public void triggerFired(Trigger trigger, JobExecutionContext context) {
        // do something with the event
    }

    public void triggerMisfired(Trigger trigger) {
        // do something with the event
    }

    public boolean vetoJobExecution(Trigger trigger, JobExecutionContext context) {
        // do something with the event
        return false;
    }

}
```

OR -

Extend TriggerListenerSupport.

```
package foo;

import org.quartz.JobExecutionContext;
import org.quartz.Trigger;
import org.quartz.listeners.TriggerListenerSupport;
```

```
public class MyOtherTriggerListener extends TriggerListenerSupport {

    private String name;

    public MyOtherTriggerListener(String name) {
        this.name = name;
    }

    public String getName() {
        return name;
    }

    @Override
    public void triggerFired(Trigger trigger, JobExecutionContext context) {
        // do something with the event
    }
}
```

Registering A TriggerListener With The Scheduler To Listen To All Triggers

```
scheduler.getListenerManager().addTriggerListener(myTriggerListener, allTriggers());
```

Registering A TriggerListener With The Scheduler To Listen To A Specific Trigger

```
scheduler.getListenerManager().addTriggerListener(myTriggerListener, triggerKeyEquals(triggerKey(
```

Registering A TriggerListener With The Scheduler To Listen To All Triggers In a Group

```
scheduler.getListenerManager().addTriggerListener(myTriggerListener, triggerGroupEquals("myTrigge
```

Contents | ‹ Prev | Next ›

How-To: Updating a trigger

Replacing a trigger

```
// Define a new Trigger
Trigger trigger = newTrigger()
    .withIdentity("newTrigger", "group1")
    .startNow()
    .build();

// tell the scheduler to remove the old trigger with the given key, and put the new one in its pl
sched.rescheduleJob(triggerKey("oldTrigger", "group1"), trigger);
```

Updating an existing trigger

```
// retrieve the trigger
Trigger oldTrigger = sched.getTrigger(triggerKey("oldTrigger", "group1");

// obtain a builder that would produce the trigger
TriggerBuilder tb = oldTrigger.getTriggerBuilder();

// update the schedule associated with the builder, and build the new trigger
// (other builder methods could be called, to change the trigger in any desired way)
Trigger newTrigger = tb.withSchedule(simpleSchedule()
    .withIntervalInSeconds(10)
    .withRepeatCount(10)
    .build();

sched.rescheduleJob(oldTrigger.getKey(), newTrigger);
```

Contents | ‹ Prev | Next ›

How-To: Trigger That Executes Every Week

If you want a trigger that always fires at a certain time of day, use CronTrigger or CalendarIntervalTrigger because they can preserve the fire time's time of day across daylight savings time changes.

Using CronTrigger

Create a CronTrigger. that executes every Wednesday at 3:00PM:

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(weeklyOnDayAndHourAndMinute(DateBuilder.WEDNESDAY, 15, 0)) // fire every wednes
    .build();
```

OR -

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startNow()
    .withSchedule(cronSchedule("0 0 15 ? * WED")) // fire every wednesday at 15:00
    .build();
```

Using SimpleTrigger

Create a SimpleTrigger that executes 3:00PM tomorrow, and then every 7 * 24 hours (which may not always be at 3:00 PM - because adding 24 hours on days where daylight savings time shifts may result in 2:00 PM or 4:00 PM depending upon whether the 3:00 PM time was started during DST or standard time):

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // first fire time 15:00:00 tomorrow
    .withSchedule(simpleSchedule()
            .withIntervalInHours(7 * 24) // interval is actually set at 7 * 24 hours' worth of mi
            .repeatForever())
    .build();
```

Using CalendarIntervalTrigger

```
trigger = newTrigger()
    .withIdentity("trigger3", "group1")
    .startAt(tomorrowAt(15, 0, 0)  // 15:00:00 tomorrow
    .withSchedule(calendarIntervalSchedule()
            .withIntervalInWeeks(1)) // interval is set in calendar weeks
    .build();
```

Configuration Reference

Configuration of Quartz is typically done through the use of a properties file, in conjunction with the use of StdSchedulerFactory (which consumes the configuration file and instantiates a scheduler).

By default, StdSchedulerFactory load a properties file named "quartz.properties" from the 'current working directory'. If that fails, then the "quartz.properties" file located (as a resource) in the org/quartz package is loaded. If you wish to use a file other than these defaults, you must define the system property 'org.quartz.properties' to point to the file you want.

Alternatively, you can explicitly initialize the factory by calling one of the initialize(xx) methods before calling getScheduler() on the StdSchedulerFactory.

Instances of the specified JobStore, ThreadPool, and other SPI classes will be created by name, and then any additional properties specified for them in the config file will be set on the instance by calling an equivalent 'set' method. For example if the properties file contains the property 'org.quartz.jobStore.myProp = 10' then after the JobStore class has been instantiated, the method 'setMyProp()' will be called on it. Type conversion to primitive Java types (int, long, float, double, boolean, and String) are performed before calling the property's setter method.

One property can reference another property's value by specifying a value following the convention of "$@other.property.name", for example, to reference the scheduler's instance name as the value for some other property, you would use "$@org.quartz.scheduler.instanceName".

The properties for configuring various aspect of a scheduler are described in these sub-documents:

Choose a topic:

1. Main Configuration (configuration of primary scheduler settings, transactions)
2. Configuration of ThreadPool (tune resources for job execution)
3. Configuration of Listeners (your application can receive notification of scheduled events)
4. Configuration of Plug-Ins (add functionality to your scheduler)
5. Configuration of RMI Server and Client (use a Quartz instance from a remote process)
6. Configuration of RAMJobStore (store jobs and triggers in memory)
7. Configuration of JDBC-JobStoreTX (store jobs and triggers in a database via JDBC)
8. Configuration of JDBC-JobStoreCMT (JDBC with JTA container-managed transactions)
9. Configuration of DataSources (for use by the JDBC-JobStores)
10. Configuration of Database Clustering (achieve fail-over and load-balancing with JDBC-JobStore)
11. Configuration of TerracottaJobStore (Clustering without a database!)

Contents | ConfigThreadPool ›

Configuration Reference

Configure Main Scheduler Settings

These properties configure the identification of the scheduler, and various other 'top level' settings.

Property Name	Req'd	Type	Default Value
org.quartz.scheduler.instanceName	no	string	'QuartzScheduler'
org.quartz.scheduler.instanceId	no	string	'NON_CLUSTERED'
org.quartz.scheduler.instanceIdGenerator.class	no	string (class name)	org.quartz.simpl .SimpleInstanceIdGenerator
org.quartz.scheduler.threadName	no	string	instanceName + '_QuartzSchedulerThread'
org.quartz.scheduler .makeSchedulerThreadDaemon	no	boolean	false
org.quartz.scheduler .threadsInheritContextClassLoaderOfInitializer	no	boolean	false
org.quartz.scheduler.idleWaitTime	no	long	30000
org.quartz.scheduler.dbFailureRetryInterval	no	long	15000
org.quartz.scheduler.classLoadHelper.class	no	string (class name)	org.quartz.simpl .CascadingClassLoadHelper
org.quartz.scheduler.jobFactory.class	no	string (class name)	org.quartz.simpl.PropertySettingJobFactory
org.quartz.context.key.SOME_KEY	no	string	none
org.quartz.scheduler.userTransactionURL	no	string (url)	'java:comp/UserTransaction'
org.quartz.scheduler .wrapJobExecutionInUserTransaction	no	boolean	false
org.quartz.scheduler.skipUpdateCheck	no	boolean	false
org.quartz.scheduler .batchTriggerAcquisitionMaxCount	no	int	1
org.quartz.scheduler .batchTriggerAcquisitionFireAheadTimeWindow	no	long	0

org.quartz.scheduler.instanceName

Can be any string, and the value has no meaning to the scheduler itself - but rather serves as a mechanism for client code to distinguish schedulers when multiple instances are used within the same program. If you are using the clustering features, you must use the same name for every instance in the cluster that is 'logically' the same Scheduler.

org.quartz.scheduler.instanceId

Can be any string, but must be unique for all schedulers working as if they are the same 'logical' Scheduler within a cluster. You may use the value "AUTO" as the instanceId if you wish the Id to be generated for you. Or the value "SYS_PROP" if you want the value to come from the system property "org.quartz.scheduler.instanceId".

org.quartz.scheduler.instanceIdGenerator.class

Only used if *org.quartz.scheduler.instanceId* is set to "AUTO". Defaults to "org.quartz.simpl.SimpleInstanceIdGenerator", which generates an instance id based upon host name and time stamp. Other IntanceIdGenerator implementations include SystemPropertyInstanceIdGenerator (which gets the instance id from the system property "org.quartz.scheduler.instanceId", and HostnameInstanceIdGenerator which uses the local host name (InetAddress.getLocalHost().getHostName()). You can also implement the InstanceIdGenerator interface your self.

org.quartz.scheduler.threadName

Can be any String that is a valid name for a java thread. If this property is not specified, the thread will receive the scheduler's name ("org.quartz.scheduler.instanceName") plus an the appended string '_QuartzSchedulerThread'.

org.quartz.scheduler.makeSchedulerThreadDaemon

A boolean value ('true' or 'false') that specifies whether the main thread of the scheduler should be a daemon thread or not. See also the *org.quartz.scheduler.makeSchedulerThreadDaemon* property for tuning the SimpleThreadPool if that is the thread pool implementation you are using (which is most likely the case).

org.quartz.scheduler.threadsInheritContextClassLoaderOfInitializer

A boolean value ('true' or 'false') that specifies whether the threads spawned by Quartz will inherit the context ClassLoader of the initializing thread (thread that initializes the Quartz instance). This will affect Quartz main scheduling thread, JDBCJobStore's misfire handling thread (if JDBCJobStore is used), cluster recovery thread (if clustering is used), and threads in SimpleThreadPool (if SimpleThreadPool is used). Setting this value to 'true' may help with class loading, JNDI look-ups, and other issues related to using Quartz within an application server.

org.quartz.scheduler.idleWaitTime

Is the amount of time in milliseconds that the scheduler will wait before re-queries for available triggers when the scheduler is otherwise idle. Normally you should not have to 'tune' this parameter, unless you're using XA transactions, and are having problems with delayed firings of triggers that should fire immediately. Values less than 5000 ms are not recommended as it will cause excessive database querying. Values less than 1000 are not legal.

org.quartz.scheduler.dbFailureRetryInterval

Is the amount of time in milliseconds that the scheduler will wait between re-tries when it has detected a loss of connectivity within the JobStore (e.g. to the database). This parameter is obviously not very meaningful when using RamJobStore.

org.quartz.scheduler.classLoadHelper.class

Defaults to the most robust approach, which is to use the "org.quartz.simpl.CascadingClassLoadHelper" class - which in turn uses every other ClassLoadHelper class until one works. You should probably not find the need to specify any other class for this property, though strange things seem to happen within application servers. All of the current possible ClassLoadHelper implementation can be found in the *org.quartz.simpl* package.

org.quartz.scheduler.jobFactory.class

The class name of the JobFactory to use. A JobFatcory is responsible for producing instances of JobClasses. The default is 'org.quartz.simpl.PropertySettingJobFactory', which simply calls newInstance() on the class to produce a new instance each time execution is about to occur. PropertySettingJobFactory also reflectively sets the job's bean properties using the contents of the SchedulerContext and Job and Trigger JobDataMaps.

org.quartz.context.key.SOME_KEY

Represent a name-value pair that will be placed into the "scheduler context" as strings. (see Scheduler.getContext()). So for example, the setting "org.quartz.context.key.MyKey = MyValue" would perform the equivalent of scheduler.getContext().put("MyKey", "MyValue").

> The Transaction-Related properties should be left out of the config file unless you are using JTA transactions.

org.quartz.scheduler.userTransactionURL

Should be set to the JNDI URL at which Quartz can locate the Application Server's UserTransaction manager. The default value (if not specified) is "java:comp/UserTransaction" - which works for almost all Application Servers. Websphere users may need to set this property to "jta/usertransaction". This is only used if Quartz is configured to use JobStoreCMT, and *org.quartz.scheduler.wrapJobExecutionInUserTransaction* is set to true.

org.quartz.scheduler.wrapJobExecutionInUserTransaction

Should be set to "true" if you want Quartz to start a UserTransaction before calling execute on your job. The Tx will commit after the job's execute method completes, and after the JobDataMap is updated (if it is a StatefulJob). The default value is "false". You may also be interested in using the *@ExecuteInJTATransaction* annotation on your job class, which lets you control for an individual job whether Quartz should start a JTA transaction - whereas this property causes it to occur for all jobs.

org.quartz.scheduler.skipUpdateCheck

Whether or not to skip running a quick web request to determine if there is an updated version of Quartz available for download. If the check runs, and an update is found, it will be reported as available in Quartz's logs. You can also disable the update check with the system property "org.terracotta.quartz.skipUpdateCheck=true" (which you can set in your system environment or as a -D on the java command line). It is recommended that you disable the update check for production deployments.

org.quartz.scheduler.batchTriggerAcquisitionMaxCount

The maximum number of triggers that a scheduler node is allowed to acquire (for firing) at once. Default value is 1. The larger the number, the more efficient firing is (in situations where there are very many triggers needing to be fired all at once) - but at the cost of possible imbalanced load between cluster nodes. If the value of this property is set to > 1, and JDBC JobStore is used, then the property

"org.quartz.jobStore.acquireTriggersWithinLock" must be set to "true" to avoid data corruption.

org.quartz.scheduler.batchTriggerAcquisitionFireAheadTimeWindow

The amount of time in milliseconds that a trigger is allowed to be acquired and fired ahead of its scheduled fire time.
Defaults to 0. The larger the number, the more likely batch acquisition of triggers to fire will be able to select and fire more than 1 trigger at a time - at the cost of trigger schedule not being honored precisely (triggers may fire this amount early). This may be useful (for performance's sake) in situations where the scheduler has very large numbers of triggers that need to be fired at or near the same time.

Contents | ‹ ConfigMain | ConfigListeners ›

Configuration Reference

Configure ThreadPool Settings

Property Name	Required	Type	Default Value
org.quartz.threadPool.class	yes	string (clas name)	null
org.quartz.threadPool.threadCount	yes	int	-1
org.quartz.threadPool.threadPriority	no	int	Thread.NORM_PRIORITY (5)

org.quartz.threadPool.class

Is the name of the ThreadPool implementation you wish to use. The threadpool that ships with Quartz is "org.quartz.simpl.SimpleThreadPool", and should meet the needs of nearly every user. It has very simple behavior and is very well tested. It provides a fixed-size pool of threads that 'live' the lifetime of the Scheduler.

org.quartz.threadPool.threadCount

Can be any positive integer, although you should realize that only numbers between 1 and 100 are very practical. This is the number of threads that are available for concurrent execution of jobs. If you only have a few jobs that fire a few times a day, then 1 thread is plenty! If you have tens of thousands of jobs, with many firing every minute, then you probably want a thread count more like 50 or 100 (this highly depends on the nature of the work that your jobs perform, and your systems resources!).

org.quartz.threadPool.threadPriority

Can be any int between *Thread.MIN_PRIORITY* (which is 1) and *Thread.MAX_PRIORITY* (which is 10). The default is *Thread.NORM_PRIORITY* (5).

SimpleThreadPool-Specific Properties

Property Name	Required	Type	Default Value
org.quartz.threadPool.makeThreadsDaemons	no	boolean	false
org.quartz.threadPool.threadsInheritGroupOfInitializingThread	no	boolean	true
org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread	no	boolean	false

org.quartz.threadPool.makeThreadsDaemons

Can be set to "true" to have the threads in the pool created as daemon threads. Default is "false". See also the *org.quartz.scheduler.makeSchedulerThreadDaemon* property.

org.quartz.threadPool.threadsInheritGroupOfInitializingThread

Can be "true" or "false", and defaults to true.

org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread

Can be "true" or "false", and defaults to false.

Custom ThreadPools

If you use your own implementation of a thread pool, you can have properties set on it reflectively simply by naming the property as thus:

Setting Properties on a Custom ThreadPool

```
org.quartz.threadPool.class = com.mycompany.goo.FooThreadPool
org.quartz.threadPool.somePropOfFooThreadPool = someValue
```

Contents | ‹ ConfigThreadPool | ConfigPlugins ›

Configuration Reference

Configure Global Listeners

Global listeners can be instantiated and configured by StdSchedulerFactory, or your application can do it itself at runtime, and then register the listeners with the scheduler. "Global" listeners listen to the events of every job/trigger rather than just the jobs/triggers that directly reference them.

Configuring listeners through the configuration file consists of giving then a name, and then specifying the class name, and any other properties to be set on the instance. The class must have a no-arg constructor, and the properties are set reflectively. Only primitive data type values (including Strings) are supported.

Thus, the general pattern for defining a "global" TriggerListener is:

Configuring a Global TriggerListener

```
org.quartz.triggerListener.NAME.class = com.foo.MyListenerClass
org.quartz.triggerListener.NAME.propName = propValue
org.quartz.triggerListener.NAME.prop2Name = prop2Value
```

And the general pattern for defining a "global" JobListener is:

Configuring a Global JobListener

```
org.quartz.jobListener.NAME.class = com.foo.MyListenerClass
org.quartz.jobListener.NAME.propName = propValue
org.quartz.jobListener.NAME.prop2Name = prop2Value
```

Contents | ‹ ConfigListeners | ConfigRMI ›

Configuration Reference

Configure Scheduler Plugins

Like listeners configuring plugins through the configuration file consists of giving then a name, and then specifying the class name, and any other properties to be set on the instance. The class must have a no-arg constructor, and the properties are set reflectively. Only primitive data type values (including Strings) are supported.

Thus, the general pattern for defining a plug-in is:

Configuring a Plugin

```
org.quartz.plugin.NAME.class = com.foo.MyPluginClass
org.quartz.plugin.NAME.propName = propValue
org.quartz.plugin.NAME.prop2Name = prop2Value
```

There are several Plugins that come with Quartz, that can be found in the *org.quartz.plugins* package (and subpackages). Example of configuring a few of them are as follows:

Sample configuration of Logging Trigger History Plugin

The logging trigger history plugin catches trigger events (it is also a trigger listener) and logs then with Jakarta Commons-Logging. See the class's JavaDoc for a list of all the possible parameters.

Sample configuration of Logging Trigger History Plugin

```
org.quartz.plugin.triggHistory.class = \
  org.quartz.plugins.history.LoggingTriggerHistoryPlugin
org.quartz.plugin.triggHistory.triggerFiredMessage = \
  Trigger \{1\}.\{0\} fired job \{6\}.\{5\} at: \{4, date, HH:mm:ss MM/dd/yyyy}
org.quartz.plugin.triggHistory.triggerCompleteMessage = \
  Trigger \{1\}.\{0\} completed firing job \{6\}.\{5\} at \{4, date, HH:mm:ss MM/dd/yyyy\}.
```

Sample configuration of XML Scheduling Data Processor Plugin

Job initialization plugin reads a set of jobs and triggers from an XML file, and adds them to the scheduler during initialization. It can also delete exiting data. See the class's JavaDoc for more details.

Sample configuration of JobInitializationPlugin

```
org.quartz.plugin.jobInitializer.class = \
  org.quartz.plugins.xml.XMLSchedulingDataProcessorPlugin
org.quartz.plugin.jobInitializer.fileNames = \
  data/my_job_data.xml
org.quartz.plugin.jobInitializer.failOnFileNotFound = true
```

The XML schema definition for the file can be found here:

http://www.quartz-scheduler.org/xml/job_scheduling_data_1_8.xsd

Sample configuration of Shutdown Hook Plugin

The shutdown-hook plugin catches the event of the JVM terminating, and calls shutdown on the scheduler.

Sample configuration of ShutdownHookPlugin

```
org.quartz.plugin.shutdownhook.class = \
  org.quartz.plugins.management.ShutdownHookPlugin
org.quartz.plugin.shutdownhook.cleanShutdown = true
```

Contents | ‹ ConfigJDBCJobStoreClustering

Configuration Reference

Configure TerracottaJobStore

TerracottaJobStore is used to store scheduling information (job, triggers and calendars) within a Terracotta server.
TerracottaJobStore is much more performant than utilizing a database for storing scheduling data (via JDBC-JobStore), and yet offers clustering features such as load-balancing and fail-over.

You may want to consider implications of how you setup your Terracotta server, particularly configuration options that turn on features such as storing data on disk, utilization of fsync, and running an array of Terracotta servers for HA.

The clustering feature works best for scaling out long-running and/or cpu-intensive jobs (distributing the work-load over multiple nodes). If you need to scale out to support thousands of short-running (e.g 1 second) jobs, consider partitioning the set of jobs by using multiple distinct schedulers. Using one scheduler currently forces the use of a cluster-wide lock, a pattern that degrades performance as you add more clients.

More information about this JobStore and Terracotta can be found at http://www.terracotta.org/quartz ›

TerracottaJobStore is selected by setting the 'org.quartz.jobStore.class' property as such:

Setting The Scheduler's JobStore to TerracottaJobStore

```
org.quartz.jobStore.class = org.terracotta.quartz.TerracottaJobStore
```

TerracottaJobStore can be tuned with the following properties:

Property Name	Required	Type	Default Value
org.quartz.jobStore.tcConfigUrl	yes	string	
org.quartz.jobStore.misfireThreshold	no	int	60000

org.quartz.jobStore.tcConfigUrl

The host and port identifying the location of the Terracotta server to connect to, such as "localhost:9510".

org.quartz.jobStore.misfireThreshold

The the number of milliseconds the scheduler will 'tolerate' a trigger to pass its next-fire-time by, before being considered "misfired". The default value (if you don't make an entry of this property in your configuration) is 60000 (60 seconds).

Contents | ‹ ConfigJobStoreCMT | ConfigJDBCJobStoreClustering ›

Configuration Reference

Configure DataSources

If you're using JDBC-Jobstore, you'll be needing a DataSource for its use (or two DataSources, if you're using JobStoreCMT).

DataSources can be configured in three ways:

1. All pool properties specified in the quartz.properties file, so that Quartz can create the DataSource itself.
2. The JNDI location of an application server managed Datasource can be specified, so that Quartz can use it.
3. Custom defined *org.quartz.utils.ConnectionProvider* implementations.

It is recommended that your Datasource max connection size be configured to be at least the number of worker threads in the thread pool plus three. You may need additional connections if your application is also making frequent calls to the scheduler API. If you are using JobStoreCMT, the "non managed" datasource should have a max connection size of at least four.

Each DataSource you define (typically one or two) must be given a name, and the properties you define for each must contain that name, as shown below. The DataSource's "NAME" can be anything you want, and has no meaning other than being able to identify it when it is assigned to the JDBCJobStore.

Quartz-created DataSources are defined with the following properties:

Property Name	Required	Type	Default Value
org.quartz.dataSource.NAME.driver	yes	String	null
org.quartz.dataSource.NAME.URL	yes	String	null
org.quartz.dataSource.NAME.user	no	String	""
org.quartz.dataSource.NAME.password	no	String	""
org.quartz.dataSource.NAME.maxConnections	no	int	10
org.quartz.dataSource.NAME.validationQuery	no	String	null
org.quartz.dataSource.NAME.idleConnectionValidationSeconds	no	int	50
org.quartz.dataSource.NAME.validateOnCheckout	no	boolean	false
org.quartz.dataSource.NAME.discardIdleConnectionsSeconds	no	int	0 (disabled)

org.quartz.dataSource.NAME.driver

Must be the java class name of the JDBC driver for your database.

org.quartz.dataSource.NAME.URL

The connection URL (host, port, etc.) for connection to your database.

org.quartz.dataSource.NAME.user

The user name to use when connecting to your database.

Quartz-created DataSources are defined with the following properties:

org.quartz.dataSource.NAME.password

The password to use when connecting to your database.

org.quartz.dataSource.NAME.maxConnections

The maximum number of connections that the DataSource can create in it's pool of connections.

org.quartz.dataSource.NAME.validationQuery

Is an optional SQL query string that the DataSource can use to detect and replace failed/corrupt connections. For example an oracle user might choose "select table_name from user_tables" - which is a query that should never fail - unless the connection is actually bad.

org.quartz.dataSource.NAME.idleConnectionValidationSeconds

The number of seconds between tests of idle connections - only enabled if the validation query property is set. Default is 50 seconds.

org.quartz.dataSource.NAME.validateOnCheckout

Whether the database sql query to validate connections should be executed every time a connection is retrieved from the pool to ensure that it is still valid. If false, then validation will occur on check-in. Default is false.

org.quartz.dataSource.NAME.discardIdleConnectionsSeconds

Discard connections after they have been idle this many seconds. 0 disables the feature. Default is 0.

Example of a Quartz-defined DataSource

```
org.quartz.dataSource.myDS.driver = oracle.jdbc.driver.OracleDriver
org.quartz.dataSource.myDS.URL = jdbc:oracle:thin:@10.0.1.23:1521:demodb
org.quartz.dataSource.myDS.user = myUser
org.quartz.dataSource.myDS.password = myPassword
org.quartz.dataSource.myDS.maxConnections = 30
```

References to Application Server DataSources are defined with the following properties:

Property Name	Required	Type	Default Value
org.quartz.dataSource.NAME.jndiURL	yes	String	null
org.quartz.dataSource.NAME.java.naming.factory.initial	no	String	null
org.quartz.dataSource.NAME.java.naming.provider.url	no	String	null
org.quartz.dataSource.NAME.java.naming.security.principal	no	String	null
org.quartz.dataSource.NAME.java.naming.security.credentials	no	String	null

org.quartz.dataSource.NAME.jndiURL

The JNDI URL for a DataSource that is managed by your application server.

References to Application Server DataSources are defined with thefollowing properties:

org.quartz.dataSource.NAME.java.naming.factory.initial

The (optional) class name of the JNDI InitialContextFactory that you wish to use.

org.quartz.dataSource.NAME.java.naming.provider.url

The (optional) URL for connecting to the JNDI context.

org.quartz.dataSource.NAME.java.naming.security.principal

The (optional) user principal for connecting to the JNDI context.

org.quartz.dataSource.NAME.java.naming.security.credentials

The (optional) user credentials for connecting to the JNDI context.

Example of a Datasource referenced from an Application Server

```
org.quartz.dataSource.myOtherDS.jndiURL=jdbc/myDataSource
org.quartz.dataSource.myOtherDS.java.naming.factory.initial=com.evermind.server.rmi.RMIInitialCon
org.quartz.dataSource.myOtherDS.java.naming.provider.url=ormi://localhost
org.quartz.dataSource.myOtherDS.java.naming.security.principal=admin
org.quartz.dataSource.myOtherDS.java.naming.security.credentials=123
```

Custom ConnectionProvider Implementations

Property Name	Required	Type	Default Value
org.quartz.dataSource.NAME.connectionProvider.class	yes	String (class name)	null

org.quartz.dataSource.NAME.connectionProvider.class

The class name of the ConnectionProvider to use. After instantiating the class, Quartz can automatically set configuration properties on the instance, bean-style.

Example of Using a Custom ConnectionProvider Implementation

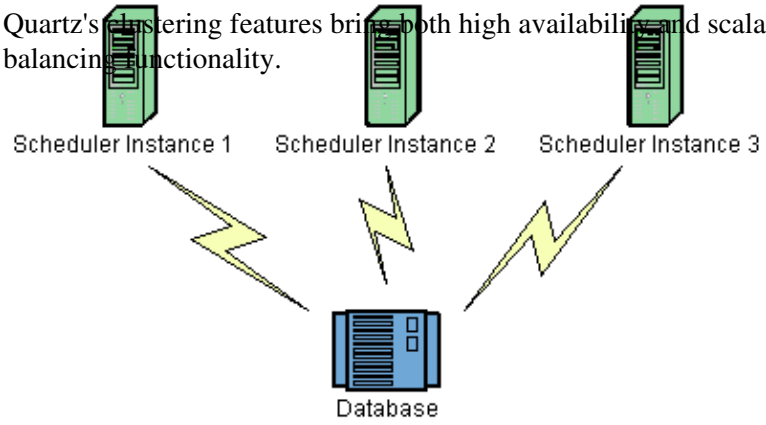
```
org.quartz.dataSource.myCustomDS.connectionProvider.class = com.foo.FooConnectionProvider
org.quartz.dataSource.myCustomDS.someStringProperty = someValue
org.quartz.dataSource.myCustomDS.someIntProperty = 5
```

Contents | ‹ ConfigDataSources | ConfigTerracottaJobStore ›

Configuration Reference

Configure Clustering with JDBC-JobStore

Quartz's clustering features brings both high availability and scalability to your scheduler via fail-over and load balancing functionality.



Clustering currently only works with the JDBC-Jobstore (JobStoreTX or JobStoreCMT), and essentially works by having each node of the cluster share the same database.

Load-balancing occurs automatically, with each node of the cluster firing jobs as quickly as it can. When a trigger's firing time occurs, the first node to acquire it (by placing a lock on it) is the node that will fire it.

Only one node will fire the job for each firing. What I mean by that is, if the job has a repeating trigger that tells it to fire every 10 seconds, then at 12:00:00 exactly one node will run the job, and at 12:00:10 exactly one node will run the job, etc. It won't necessarily be the same node each time - it will more or less be random which node runs it. The load balancing mechanism is near-random for busy schedulers (lots of triggers) but favors the same node for non-busy (e.g. few triggers) schedulers.

Fail-over occurs when one of the nodes fails while in the midst of executing one or more jobs. When a node fails, the other nodes detect the condition and identify the jobs in the database that were in progress within the failed node.
Any jobs marked for recovery (with the "requests recovery" property on the JobDetail) will be re-executed by the remaining nodes. Jobs not marked for recovery will simply be freed up for execution at the next time a related trigger fires.

The clustering feature works best for scaling out long-running and/or cpu-intensive jobs (distributing the work-load over multiple nodes). If you need to scale out to support thousands of short-running (e.g 1 second) jobs, consider partitioning the set of jobs by using multiple distinct schedulers (including multiple clustered schedulers for HA). The scheduler makes use of a cluster-wide lock, a pattern that degrades performance as you add more nodes (when going beyond about three nodes - depending upon your database's capabilities, etc.).

Enable clustering by setting the "org.quartz.jobStore.isClustered" property to "true". Each instance in the cluster should use the same copy of the quartz.properties file. Exceptions of this would be to use properties files that are identical, with the following allowable exceptions: Different thread pool size, and different value

for the "org.quartz.scheduler.instanceId" property. Each node in the cluster MUST have a unique instanceId, which is easily done (without needing different properties files) by placing "AUTO" as the value of this property. See the info about the configuration properties of JDBC-JobStore for more information.

> Never run clustering on separate machines, unless their clocks are synchronized using some form of time-sync service (daemon) that runs very regularly (the clocks must be within a second of each other). See http://www.boulder.nist.gov/timefreq/service/its.htm if you are unfamiliar with how to do this.
>
> Never start (scheduler.start()) a non-clustered instance against the same set of database tables that any other instance is running (start()ed) against. You may get serious data corruption, and will definitely experience erratic behavior.

Example Properties For A Clustered Scheduler

```
#============================================================================
# Configure Main Scheduler Properties
#============================================================================

org.quartz.scheduler.instanceName = MyClusteredScheduler
org.quartz.scheduler.instanceId = AUTO

#============================================================================
# Configure ThreadPool
#============================================================================

org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 25
org.quartz.threadPool.threadPriority = 5

#============================================================================
# Configure JobStore
#============================================================================

org.quartz.jobStore.misfireThreshold = 60000

org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
org.quartz.jobStore.driverDelegateClass = org.quartz.impl.jdbcjobstore.oracle.OracleDelegate
org.quartz.jobStore.useProperties = false
org.quartz.jobStore.dataSource = myDS
org.quartz.jobStore.tablePrefix = QRTZ_

org.quartz.jobStore.isClustered = true
org.quartz.jobStore.clusterCheckinInterval = 20000

#============================================================================
# Configure Datasources
#============================================================================

org.quartz.dataSource.myDS.driver = oracle.jdbc.driver.OracleDriver
org.quartz.dataSource.myDS.URL = jdbc:oracle:thin:@polarbear:1521:dev
org.quartz.dataSource.myDS.user = quartz
org.quartz.dataSource.myDS.password = quartz
org.quartz.dataSource.myDS.maxConnections = 5
org.quartz.dataSource.myDS.validationQuery=select 0 from dual
```

Contents | ‹ ConfigJobStoreTX | ConfigDataSources ›

Configuration Reference

Configure JDBC-JobStoreCMT

JDBCJobStore is used to store scheduling information (job, triggers and calendars) within a relational database. There are actually two seperate JDBCJobStore classes that you can select between, depending on the transactional behaviour you need.

JobStoreCMT relies upon transactions being managed by the application which is using Quartz. A JTA transaction must be in progress before attempt to schedule (or unschedule) jobs/triggers. This allows the "work" of scheduling to be part of the applications "larger" transaction. JobStoreCMT actually requires the use of two datasources - one that has it's connection's transactions managed by the application server (via JTA) and one datasource that has connections that do not participate in global (JTA) transactions. JobStoreCMT is appropriate when applications are using JTA transactions (such as via EJB Session Beans) to perform their work.

The JobStore is selected by setting the 'org.quartz.jobStore.class' property as such:

Setting The Scheduler's JobStore to JobStoreCMT

```
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreCMT
```

JobStoreCMT can be tuned with the following properties:

Property Name	Required	Type	Default Value
org.quartz.jobStore.driverDelegateClass	yes	string	null
org.quartz.jobStore.dataSource	yes	string	null
org.quartz.jobStore.nonManagedTXDataSource	yes	string	null
org.quartz.jobStore.tablePrefix	no	string	"QRTZ_"
org.quartz.jobStore.useProperties	no	boolean	false
org.quartz.jobStore.misfireThreshold	no	int	60000
org.quartz.jobStore.isClustered	no	boolean	false
org.quartz.jobStore.clusterCheckinInterval	no	long	15000
org.quartz.jobStore.maxMisfiresToHandleAtATime	no	int	20
org.quartz.jobStore.dontSetAutoCommitFalse	no	boolean	false
org.quartz.jobStore.dontSetNonManagedTXConnectionAutoCommitFalse	no	boolean	false
org.quartz.jobStore.selectWithLockSQL	no	string	"SELECT * FROM {0}LOCKS WHERE SCHED_NAME = {1} AND LOCK_NAME = ? FOR UPDATE"
org.quartz.jobStore.txIsolationLevelSerializable	no	boolean	false

org.quartz.jobStore.txIsolationLevelReadCommitted	no	boolean	false
org.quartz.jobStore.acquireTriggersWithinLock	no	boolean	false (or true - see doc below)
org.quartz.jobStore.lockHandler.class	no	string	null
org.quartz.jobStore.driverDelegateInitString	no	string	null

org.quartz.jobStore.driverDelegateClass

Driver delegates understand the particular 'dialects' of varies database systems. Possible choices include:

- org.quartz.impl.jdbcjobstore.StdJDBCDelegate (for fully JDBC-compliant drivers)
- org.quartz.impl.jdbcjobstore.MSSQLDelegate (for Microsoft SQL Server, and Sybase)
- org.quartz.impl.jdbcjobstore.PostgreSQLDelegate
- org.quartz.impl.jdbcjobstore.WebLogicDelegate (for WebLogic drivers)
- org.quartz.impl.jdbcjobstore.oracle.OracleDelegate
- org.quartz.impl.jdbcjobstore.oracle.WebLogicOracleDelegate (for Oracle drivers used within Weblogic)
- org.quartz.impl.jdbcjobstore.oracle.weblogic.WebLogicOracleDelegate (for Oracle drivers used within Weblogic)
- org.quartz.impl.jdbcjobstore.CloudscapeDelegate
- org.quartz.impl.jdbcjobstore.DB2v6Delegate
- org.quartz.impl.jdbcjobstore.DB2v7Delegate
- org.quartz.impl.jdbcjobstore.DB2v8Delegate
- org.quartz.impl.jdbcjobstore.HSQLDBDelegate
- org.quartz.impl.jdbcjobstore.PointbaseDelegate
- org.quartz.impl.jdbcjobstore.SybaseDelegate

Note that many databases are known to work with the StdJDBCDelegate, while others are known to work with delegates for other databases, for example Derby works well with the Cloudscape delegate (no surprise there).

org.quartz.jobStore.dataSource

The value of this property must be the name of one the DataSources defined in the configuration properties file. For JobStoreCMT, it is required that this DataSource contains connections that are capable of participating in JTA (e.g. container-managed) transactions. This typically means that the DataSource will be configured and maintained within and by the application server, and Quartz will obtain a handle to it via JNDI. See the configuration docs for DataSources for more information.

org.quartz.jobStore.nonManagedTXDataSource

JobStoreCMT *requires* a (second) datasource that contains connections that will *not* be part of container-managed transactions. The value of this property must be the name of one the DataSources defined in the configuration properties file. This datasource must contain non-CMT connections, or in other words, connections for which it is legal for Quartz to directly call commit() and rollback() on.

org.quartz.jobStore.tablePrefix

JDBCJobStore's "table prefix" property is a string equal to the prefix given to Quartz's tables that were created in your database. You can have multiple sets of Quartz's tables within the same database if they use different table prefixes.

org.quartz.jobStore.useProperties

The "use properties" flag instructs JDBCJobStore that all values in JobDataMaps will be Strings, and therefore can be stored as name-value pairs, rather than storing more complex objects in their serialized form in the BLOB column. This is can be handy, as you avoid the class versioning issues that can arise from serializing your non-String classes into a BLOB.

org.quartz.jobStore.misfireThreshold

The the number of milliseconds the scheduler will 'tolerate' a trigger to pass its next-fire-time by, before being considered "misfired". The default value (if you don't make an entry of this property in your configuration) is 60000 (60 seconds).

org.quartz.jobStore.isClustered

Set to "true" in order to turn on clustering features. This property must be set to "true" if you are having multiple instances of Quartz use the same set of database tables... otherwise you will experience havoc. See the configuration docs for clustering for more information.

org.quartz.jobStore.clusterCheckinInterval

Set the frequency (in milliseconds) at which this instance "checks-in"* with the other instances of the cluster. Affects the quickness of detecting failed instances.

org.quartz.jobStore.maxMisfiresToHandleAtATime

The maximum number of misfired triggers the jobstore will handle in a given pass. Handling many (more than a couple dozen) at once can cause the database tables to be locked long enough that the performance of firing other (not yet misfired) triggers may be hampered.

org.quartz.jobStore.dontSetAutoCommitFalse

Setting this parameter to "true" tells Quartz not to call *setAutoCommit(false)* on connections obtained from the DataSource(s). This can be helpful in a few situations, such as if you have a driver that complains if it is called when it is already off. This property defaults to false, because most drivers require that *setAutoCommit(false)* is called.

org.quartz.jobStore.dontSetNonManagedTXConnectionAutoCommitFalse

The same as the property *org.quartz.jobStore.dontSetAutoCommitFalse*, except that it applies to the nonManagedTXDataSource.

org.quartz.jobStore.selectWithLockSQL

Must be a SQL string that selects a row in the "LOCKS" table and places a lock on the row. If not set, the default is "SELECT * FROM {0}LOCKS WHERE SCHED_NAME = {1} AND LOCK_NAME = ? FOR UPDATE", which works for most databases. The "{0}" is replaced during run-time with the TABLE_PREFIX that you configured above. The "{1}" is replaced with the scheduler's name.

org.quartz.jobStore.txIsolationLevelSerializable

A value of "true" tells Quartz to call *setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE)* on JDBC connections. This can be helpful to prevent lock timeouts with some databases under high load, and "long-lasting" transactions.

org.quartz.jobStore.txIsolationLevelReadCommitted

When set to "true", this property tells Quartz to call *setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED)* on the non-managed JDBC connections. This can be helpful to prevent lock timeouts with some databases (such as DB2) under high load, and "long-lasting" transactions.

org.quartz.jobStore.acquireTriggersWithinLock

Whether or not the acquisition of next triggers to fire should occur within an explicit database lock. This was once necessary (in previous versions of Quartz) to avoid dead-locks with particular databases, but is no longer considered necessary, hence the default value is "false".

If "org.quartz.scheduler.batchTriggerAcquisitionMaxCount" is set to > 1, and JDBC JobStore is used, then this property must be set to "true" to avoid data corruption (as of Quartz 2.1.1 "true" is now the default if batchTriggerAcquisitionMaxCount is set > 1).

org.quartz.jobStore.lockHandler.class

The class name to be used to produce an instance of a org.quartz.impl.jdbcjobstore.Semaphore to be used for locking control on the job store data. This is an advanced configuration feature, which should not be used by most users. By default, Quartz will select the most appropriate (pre-bundled) Semaphore implementation to use. "org.quartz.impl.jdbcjobstore.UpdateLockRowSemaphore" QUARTZ-497 may be of interest to MS SQL Server users. "JTANonClusteredSemaphore" which is bundled with Quartz may give improved performance when using JobStoreCMT, though it is an experimental implementation. See QUARTZ-441 and QUARTZ-442

org.quartz.jobStore.driverDelegateInitString

A pipe-delimited list of properties (and their values) that can be passed to the DriverDelegate during initialization time.

The format of the string is as such:

```
"settingName=settingValue|otherSettingName=otherSettingValue|..."
```

The StdJDBCDelegate and all of its descendants (all delegates that ship with Quartz) support a property called 'triggerPersistenceDelegateClasses' which can be set to a comma-separated list of classes that implement the TriggerPersistenceDelegate interface for storing custom trigger types. See the Java classes SimplePropertiesTriggerPersistenceDelegateSupport and SimplePropertiesTriggerPersistenceDelegateSupport for examples of writing a persistence delegate for a custom trigger.

Contents | ‹ ConfigRAMJobStore | ConfigJobStoreCMT ›

Configuration Reference

Configure JDBC-JobStoreTX

JDBCJobStore is used to store scheduling information (job, triggers and calendars) within a relational database. There are actually two seperate JDBCJobStore classes that you can select between, depending on the transactional behaviour you need.

JobStoreTX manages all transactions itself by calling commit() (or rollback()) on the database connection after every action (such as the addition of a job). JDBCJobStore is appropriate if you are using Quartz in a stand-alone application, or within a servlet container if the application is not using JTA transactions.

The JobStoreTX is selected by setting the 'org.quartz.jobStore.class' property as such:

Setting The Scheduler's JobStore to JobStoreTX

```
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
```

JobStoreTX can be tuned with the following properties:

Property Name	Required	Type	Default Value
org.quartz.jobStore.driverDelegateClass	yes	string	null
org.quartz.jobStore.dataSource	yes	string	null
org.quartz.jobStore.tablePrefix	no	string	"QRTZ_"
org.quartz.jobStore.useProperties	no	boolean	false
org.quartz.jobStore.misfireThreshold	no	int	60000
org.quartz.jobStore.isClustered	no	boolean	false
org.quartz.jobStore.clusterCheckinInterval	no	long	15000
org.quartz.jobStore.maxMisfiresToHandleAtATime	no	int	20
org.quartz.jobStore.dontSetAutoCommitFalse	no	boolean	false
org.quartz.jobStore.selectWithLockSQL	no	string	"SELECT * FROM {0}LOCKS WHERE SCHED_NAME = {1} AND LOCK_NAME = ? FOR UPDATE"
org.quartz.jobStore.txIsolationLevelSerializable	no	boolean	false
org.quartz.jobStore.acquireTriggersWithinLock	no	boolean	false (or true - see doc below)
org.quartz.jobStore.lockHandler.class	no	string	null
org.quartz.jobStore.driverDelegateInitString	no	string	null

org.quartz.jobStore.driverDelegateClass

Driver delegates understand the particular 'dialects' of varies database systems. Possible choices include:

- org.quartz.impl.jdbcjobstore.StdJDBCDelegate (for fully JDBC-compliant drivers)
- org.quartz.impl.jdbcjobstore.MSSQLDelegate (for Microsoft SQL Server, and Sybase)
- org.quartz.impl.jdbcjobstore.PostgreSQLDelegate
- org.quartz.impl.jdbcjobstore.WebLogicDelegate (for WebLogic drivers)

- org.quartz.impl.jdbcjobstore.oracle.OracleDelegate
- org.quartz.impl.jdbcjobstore.oracle.WebLogicOracleDelegate (for Oracle drivers used within Weblogic)
- org.quartz.impl.jdbcjobstore.oracle.weblogic.WebLogicOracleDelegate (for Oracle drivers used within Weblogic)
- org.quartz.impl.jdbcjobstore.CloudscapeDelegate
- org.quartz.impl.jdbcjobstore.DB2v6Delegate
- org.quartz.impl.jdbcjobstore.DB2v7Delegate
- org.quartz.impl.jdbcjobstore.DB2v8Delegate
- org.quartz.impl.jdbcjobstore.HSQLDBDelegate
- org.quartz.impl.jdbcjobstore.PointbaseDelegate
- org.quartz.impl.jdbcjobstore.SybaseDelegate

Note that many databases are known to work with the StdJDBCDelegate, while others are known to work with delegates for other databases, for example Derby works well with the Cloudscape delegate (no surprise there).

org.quartz.jobStore.dataSource

The value of this property must be the name of one the DataSources defined in the configuration properties file. See the configuration docs for DataSources for more information.

org.quartz.jobStore.tablePrefix

JDBCJobStore's "table prefix" property is a string equal to the prefix given to Quartz's tables that were created in your database. You can have multiple sets of Quartz's tables within the same database if they use different table prefixes.

org.quartz.jobStore.useProperties

The "use properties" flag instructs JDBCJobStore that all values in JobDataMaps will be Strings, and therefore can be stored as name-value pairs, rather than storing more complex objects in their serialized form in the BLOB column. This is can be handy, as you avoid the class versioning issues that can arise from serializing your non-String classes into a BLOB.

org.quartz.jobStore.misfireThreshold

The the number of milliseconds the scheduler will 'tolerate' a trigger to pass its next-fire-time by, before being considered "misfired". The default value (if you don't make an entry of this property in your configuration) is 60000 (60 seconds).

org.quartz.jobStore.isClustered

Set to "true" in order to turn on clustering features. This property must be set to "true" if you are having multiple instances of Quartz use the same set of database tables... otherwise you will experience havoc. See the configuration docs for clustering for more information.

org.quartz.jobStore.clusterCheckinInterval

Set the frequency (in milliseconds) at which this instance "checks-in"* with the other instances of the cluster. Affects the quickness of detecting failed instances.

org.quartz.jobStore.maxMisfiresToHandleAtATime

The maximum number of misfired triggers the jobstore will handle in a given pass. Handling many (more than a couple dozen) at once can cause the database tables to be locked long enough that the performance of firing other (not yet misfired) triggers may be hampered.

org.quartz.jobStore.dontSetAutoCommitFalse

Setting this parameter to "true" tells Quartz not to call setAutoCommit(false) on connections obtained from the DataSource(s). This can be helpful in a few situations, such as if you have a driver that complains if it is called when it is already off. This property defaults to false, because most drivers require that setAutoCommit(false) is called.

org.quartz.jobStore.selectWithLockSQL

Must be a SQL string that selects a row in the "LOCKS" table and places a lock on the row. If not set, the default is "SELECT * FROM {0}LOCKS WHERE SCHED_NAME = {1} AND LOCK_NAME = ? FOR UPDATE", which works for most databases. The "{0}" is replaced during run-time with the TABLE_PREFIX that you configured above. The "{1}" is replaced with the scheduler's name.

org.quartz.jobStore.txIsolationLevelSerializable

A value of "true" tells Quartz (when using JobStoreTX or CMT) to call setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE) on JDBC connections. This can be helpful to prevent lock timeouts with some databases under high load, and "long-lasting" transactions.

org.quartz.jobStore.acquireTriggersWithinLock

Whether or not the acquisition of next triggers to fire should occur within an explicit database lock. This was once necessary (in previous versions of Quartz) to avoid dead-locks with particular databases, but is no longer considered necessary, hence the default value is "false".

If "org.quartz.scheduler.batchTriggerAcquisitionMaxCount" is set to > 1, and JDBC JobStore is used, then this property must be set to "true" to avoid data corruption (as of Quartz 2.1.1 "true" is now the default if batchTriggerAcquisitionMaxCount is set > 1).

org.quartz.jobStore.lockHandler.class

The class name to be used to produce an instance of a org.quartz.impl.jdbcjobstore.Semaphore to be used for locking control on the job store data. This is an advanced configuration feature, which should not be used by most users. By default, Quartz will select the most appropriate (pre-bundled) Semaphore implementation to use. "org.quartz.impl.jdbcjobstore.UpdateLockRowSemaphore" QUARTZ-497 may be of interest to MS SQL Server users. See QUARTZ-441.

org.quartz.jobStore.driverDelegateInitString

A pipe-delimited list of properties (and their values) that can be passed to the DriverDelegate during initialization time.

The format of the string is as such:

```
"settingName=settingValue|otherSettingName=otherSettingValue|..."
```

The StdJDBCDelegate and all of its descendants (all delegates that ship with Quartz) support a property called 'triggerPersistenceDelegateClasses' which can be set to a comma-separated list of classes that implement the TriggerPersistenceDelegate interface for storing custom trigger types. See the Java classes SimplePropertiesTriggerPersistenceDelegateSupport and SimplePropertiesTriggerPersistenceDelegateSupport for examples of writing a persistence delegate for a custom trigger.

Contents | ‹ ConfigRMI | ConfigJobStoreTX ›

Configuration Reference

Configure RAMJobStore

RAMJobStore is used to store scheduling information (job, triggers and calendars) within memory. RAMJobStore is fast and lightweight, but all scheduling information is lost when the process terminates.

RAMJobStore is selected by setting the 'org.quartz.jobStore.class' property as such:

Setting The Scheduler's JobStore to RAMJobStore

```
org.quartz.jobStore.class = org.quartz.simpl.RAMJobStore
```

RAMJobStore can be tuned with the following properties:

Property Name	Required	Type	Default Value
org.quartz.jobStore.misfireThreshold	no	int	60000

org.quartz.jobStore.misfireThreshold

The the number of milliseconds the scheduler will 'tolerate' a trigger to pass its next-fire-time by, before being considered "misfired". The default value (if you don't make an entry of this property in your configuration) is 60000 (60 seconds).

Contents | ‹ ConfigPlugins | ConfigRAMJobStore ›

Configuration Reference

Configure Scheduler RMI Settings

None of the primary properties are required, and all have 'reasonable' defaults. When using Quartz via RMI, you need to start an instance of Quartz with it configured to "export" its services via RMI. You then create clients to the server by configuring a Quartz scheduler to "proxy" its work to the server.

> Some users experience problems with class availability (namely Job classes) between the client and server. To work through these problems you'll need an understanding of RMI's "codebase" and RMI security managers. You may find these resources to be useful:

An excellent description of RMI and codebase: http://www.kedwards.com/jini/codebase.html . One of the important points is to realize that "codebase" is used by the client!

Quick info about security managers:
http://gethelp.devx.com/techtips/java_pro/10MinuteSolutions/10min0500.asp

And finally from the java API docs, read the docs for the RMISecurityManager

http://java.sun.com/j2se/1.4.2/docs/api/java/rmi/RMISecurityManager.html

Property Name	Required	Default Value
org.quartz.scheduler.rmi.export	no	false
org.quartz.scheduler.rmi.registryHost	no	'localhost'
org.quartz.scheduler.rmi.registryPort	no	1099
org.quartz.scheduler.rmi.createRegistry	no	'never'
org.quartz.scheduler.rmi.serverPort	no	random
org.quartz.scheduler.rmi.proxy	no	false

org.quartz.scheduler.rmi.export

If you want the Quartz Scheduler to export itself via RMI as a server then set the 'rmi.export' flag to true.

org.quartz.scheduler.rmi.registryHost

The host at which the RMI Registry can be found (often 'localhost').

org.quartz.scheduler.rmi.registryPort

The port on which the RMI Registry is listening (usually 1099).

org.quartz.scheduler.rmi.createRegistry

Set the 'rmi.createRegistry' flag according to how you want Quartz to cause the creation of an RMI Registry. Use "false" or "never" if you don't want Quartz to create a registry (e.g. if you already have an external registry running). Use "true" or "as_needed" if you want Quartz to first attempt to use an existing registry, and then fall back to creating one. Use "always" if you want Quartz to attempt creating a Registry, and then fall back to using an existing one. If a registry is created, it will be bound to port number in the given

'org.quartz.scheduler.rmi.registryPort' property, and 'org.quartz.rmi.registryHost' should be "localhost".

org.quartz.scheduler.rmi.serverPort

The port on which the the Quartz Scheduler service will bind and listen for connections. By default, the RMI service will 'randomly' select a port as the scheduler is bound to the RMI Registry.

org.quartz.scheduler.rmi.proxy

If you want to connect to (use) a remotely served scheduler, then set the 'org.quartz.scheduler.rmi.proxy' flag to true. You must also then specify a host and port for the RMI Registry process - which is typically 'localhost' port 1099.

> It does not make sense to specify a 'true' value for both 'org.quartz.scheduler.rmi.export' and 'org.quartz.scheduler.rmi.proxy' in the same config file - if you do, the 'export' option will be ignored. A value of 'false' for both 'export' and 'proxy' properties is of course valid, if you're not using Quartz via RMI.

Best Practices

Production System Tips

Skip Update Check

Quartz contains an "update check" feature that connects to a server to check if there is a new version of Quartz available for download. This check runs asynchronously and does not affect startup/initialization time of Quartz, and it fails gracefully if the connection cannot be made. If the check runs, and an update is found, it will be reported as available in Quartz's logs.

You can disable the update check with the Quartz config property "org.quartz.scheduler.skipUpdateCheck: true" or the system property "org.terracotta.quartz.skipUpdateCheck=true" (which you can set in your system environment or as a -D on the java command line). It is recommended that you disable the update check for production deployments.

JobDataMap Tips

Only Store Primitive Data Types (including Strings) In the JobDataMap

Only store primitive data types (including Strings) in JobDataMap to avoid data serialization issues short and long-term.

Use the Merged JobDataMap

The *JobDataMap* that is found on the *JobExecutionContext* during *Job* execution serves as a convenience. It is a merge of the *JobDataMap* found on the *JobDetail* and the one found on the *Trigger*, with the value in the latter overriding any same-named values in the former.

Storing *JobDataMap* values on a *Trigger* can be useful in the case where you have a *Job* that is stored in the scheduler for regular/repeated use by multiple Triggers, yet with each independent triggering, you want to supply the Job with different data inputs.

In light of all of the above, we recommend as a best practice the following: Code within the Job.execute(..) method should generally retrieve values from the *JobDataMap* on found on the *JobExecutionContext*, rather than directly from the one on the *JobDetail*.

Trigger Tips

Use TriggerUtils

TriggerUtils:

- Offers a simpler way to create triggers (schedules)
- Has various methods for creating triggers with schedules that meet particular descriptions, as opposed to directly instantiating triggers of a specific type (i.e. SimpleTrigger, CronTrigger, etc.) and then invoking various setter methods to configure them

- Offers a simple way to create Dates (for start/end dates)
- Offers helpers for analyzing triggers (e.g. calculating future fire times)

JDBC JobStore

Never Write Directly To Quartz's Tables

Writing scheduling data directly to the database (via SQL) rather than using scheduling API:

- Results in data corruption (deleted data, scrambled data)
- Results in job seemingly "vanishing" without executing when a trigger's fire time arrives
- Results in job not executing "just sitting there" when a trigger's fire time arrives
- May result in: Dead-locks
- Other strange problems and data corruption

Never Point A Non-Clustered Scheduler At the Same Database As Another Scheduler With The Same Scheduler Name

If you point more than one scheduler instance at the same set of database tables, and one or more of those instances is not configured for clustering, any of the following may occur:

- Results in data corruption (deleted data, scrambled data)
- Results in job seemingly "vanishing" without executing when a trigger's fire time arrives
- Results in job not executing, "just sitting there" when a trigger's fire time arrives
- May result in: Dead-locks
- Other strange problems and data corruption

Ensure Adequate Datasource Connection Size

It is recommended that your Datasource max connection size be configured to be at least the number of worker threads in the thread pool plus three. You may need additional connections if your application is also making frequent calls to the scheduler API. If you are using JobStoreCMT, the "non managed" datasource should have a max connection size of at least four.

Daylight Savings Time

Avoid Scheduling Jobs Near the Transition Hours of Daylight Savings Time

NOTE: Specifics of the transition hour and the amount of time the clock moves forward or back varies by locale see: https://secure.wikimedia.org/wikipedia/en/wiki/Daylight_saving_time_around_the_world.

SimpleTriggers are not affected by Daylight Savings Time as they always fire at an exact millisecond in time, and repeat an exact number of milliseconds apart.

Because CronTriggers fire at given hours/minutes/seconds, they are subject to some oddities when DST transitions occur.

As an example of possible issues, scheduling in the United States within TimeZones/locations that observe Daylight Savings time, the following problems may occur if using CronTrigger and scheduling fire times during the hours of 1:00 AM and 2:00 AM:

- 1:05 AM may occur twice! - duplicate firings on CronTrigger possible
- 2:05 AM may never occur! - missed firings on CronTrigger possible

Again, specifics of time and amount of adjustment varies by locale.

Other trigger types that are based on sliding along a calendar (rather than exact amounts of time), such as CalenderIntervalTrigger, will be similarly affected - but rather than missing a firing, or firing twice, may end up having it's fire time shifted by an hour.

Jobs

Waiting For Conditions

Long-running jobs prevent others from running (if all threads in the ThreadPool are busy).

If you feel the need to call Thread.sleep() on the worker thread executing the Job, it is typically a sign that the job is not ready to do the rest of its work because it needs to wait for some condition (such as the availability of a data record) to become true.

A better solution is to release the worker thread (exit the job) and allow other jobs to execute on that thread. *The job can reschedule itself, or other jobs before it exits.*

Throwing Exceptions

A Job's execute method should contain a try-catch block that handles all possible exceptions.

If a job throws an exception, Quartz will typically immediately re-execute it (and it will likely throw the same exception again). It's better if the job catches all exception it may encounter, handle them, and reschedule itself, or other jobs. to work around the issue.

Recoverability and Idempotence

In-progress Jobs marked "recoverable" are automatically re-executed after a scheduler fails. This means some of the job's "work" will be executed twice.

This means the job should be coded in such a way that its work is idempotent.

Listeners (TriggerListener, JobListener, SchedulerListener

Keep Code In Listeners Concise And Efficient

Performing large amounts of work is discouraged, as the thread that would be executing the job (or completing the trigger and moving on to firing another job, etc.) will be tied up within the listener.

Handle Exceptions

Every listener method should contain a try-catch block that handles all possible exceptions.

If a listener throws an exception, it may cause other listeners not to be notified and/or prevent the execution of the job, etc.

Exposing Scheduler Functionality Through Applications

Be Careful of Security!

Some users expose Quartz's Scheduler functionality through an application user interface. This can be very useful, though it can also be extremely dangerous.

Be sure you don't mistakenly allow users to define jobs of any type they wish, with whatever parameters they wish. For example, Quartz ships with a pre-made job *org.quartz.jobs.NativeJob*, which will execute any arbitrary native (operating system) system command that it is defined to. Malicious users could use this to take control of, or destroy your system.

Likewise other jobs such as SendEmailJob, and virtually any others could be used for malicious intent.

Allowing users to define whatever job they want effectively opens your system to all sorts of vulnerabilities comparable/equivalent to Command Injection Attacks as defined by OWASP and MITRE.

Frequently Answered Questions about Quartz

General Questions:

Miscellaneous Questions:

Questions about Jobs:

Questions about Triggers:

Questions about JDBCJobStore:

Questions about Transactions:

Questions about Clustering, (Scaling and High-Availability) Features:

Questions about Spring:

General Questions

What is Quartz?

Quartz is a job scheduling system that can be integrated with, or used along side virtually any other software system. The term "job scheduler" seems to conjure different ideas for different people. As you read this tutorial, you should be able to get a firm idea of what we mean when we use this term, but in short, a job scheduler is a system that is responsible for executing (or notifying) other software components when a pre-determined (scheduled) time arrives.

Quartz is quite flexible, and contains multiple usage paradigms that can be used separately or together, in order to achieve your desired behavior, and enable you to write your code in the manner that seems most 'natural' to your project.

Quartz is very light-weight, and requires very little setup/configuration - it can actually be used 'out-of-the-box' if your needs are relatively basic.

Quartz is fault-tolerant, and can persist ('remember') your scheduled jobs between system restarts.

Although Quartz is extremely useful for simply running certain system processes on given schedules, the full potential of Quartz can be realized when you learn how to use it to drive the flow of your application's business processes.

What is Quartz - From a Software Component View?

Quartz is distributed as a small java library (.jar file) that contains all of the core Quartz functionality. The main interface (API) to this functionality is the Scheduler interface. It provides simple operations such as scheduling/unscheduling jobs, starting/stopping/pausing the scheduler.

If you wish to schedule your own software components for execution they must implement the simple Job interface, which contains the method execute(). If you wish to have components notified when a scheduled fire-time arrives, then the components should implement either the TriggerListener or JobListener interface.

The main Quartz 'process' can be started and ran within your own application, as a stand-alone application (with an RMI interface), or within a J2EE app. server to be used as a resource by your J2EE components.

Why not just use java.util.Timer?

Since JDK 1.3, Java has "built-in" timer capabilities, through the java.util.Timer and java.util.TimerTask classes - why would someone use Quartz rather than these standard features?

There are many reasons! Here are a few:

1. Timers have no persistence mechanism.
2. Timers have inflexible scheduling (only able to set start-time & repeat interval, nothing based on dates, time of day, etc.)
3. Timers don't utilize a thread-pool (one thread per timer)
4. Timers have no real management schemes - you'd have to write your own mechanism for being able to remember, organize and retrieve your tasks by name, etc.

...of course to some simple applications these features may not be important, in which case it may then be the right decision not to use Quartz.

What is Terracotta's involvement with Quartz?

Terracotta acquired the rights to the Quartz Scheduler project in November, 2009. Terracotta and the original Quartz team together can provide improved features and services for Quartz. Quartz is still (and always will be) an open source project with contributors from around the world. Terracotta brings additional expertise for improving some of the advanced features of Quartz, and offers commercial support services to Quartz users.

Are there commercial support services available for Quartz?

Yes, Terracotta offers great commercial support services to Quartz users.

What are the available alternatives to Quartz?

There are no known competing open source projects (there are a few other open source schedulers, but they are basically just Cron replacements written in Java).

Commercially, you will want to look at the well-regarded Flux scheduler.

How do I build the Quartz source?

Although Quartz ships "pre-built" many people like to make their own alterations and/or build the latest 'non-released' version of Quartz from CVS. To do this, follow the instructions in the "README.TXT" file that ships with Quartz.

How do I disable the update check?

Quartz contains an "update check" feature that connects to a server to check if there is a new version of Quartz available for download. This check runs asynchronously and does not affect startup/initialization time of Quartz, and it fails gracefully if the connection cannot be made. If the check runs, and an update is found, it will be reported as available in Quartz's logs.

You can disable the update check with the Quartz config property "org.quartz.scheduler.skipUpdateCheck: true" or the system property "org.terracotta.quartz.skipUpdateCheck=true" (which you can set in your system environment or as a -D on the java command line). It is recommended that you disable the update check for production deployments.

Miscellaneous Questions

How many jobs is Quartz capable of running?

This is a tough question to answer... the answer is basically "it depends".

I know you hate that answer, so here's some information about what it depends "on".

First off, the JobStore that you use plays a significant factor. The RAM-based JobStore is MUCH (1000x) faster than the JDBC-based JobStore. The speed of JDBC-JobStore depends almost entirely on the speed of

the connection to your database, which data base system that you use, and what hardware the database is running on. Quartz actually does very little processing itself, nearly all of the time is spent in the database. Of course RAMJobStore has a more finite limit on how many Jobs and Triggers can be stored, as you're sure to have less RAM than hard-drive space for a database. You may also look at the FAQ "How do I improve the performance of JDBC-JobStore?"

So, the limiting factor of the number of Triggers and Jobs Quartz can "store" and monitor is really the amount of storage space available to the JobStore (either the amount of RAM or the amount of disk space).

Now, aside from "how many can I store?" is the question of "how many jobs can Quartz be running at the same moment in time?"

One thing that CAN slow down quartz itself is using a lot of listeners (TriggerListeners, JobListeners, and SchedulerListeners). The time spent in each listener obviously adds into the time spent "processing" a job's execution, outside of actual execution of the job. This doesn't mean that you should be terrified of using listeners, it just means that you should use them judiciously - don't create a bunch of "global" listeners if you can really make more specialized ones. Also don't do "expensive" things in the listeners, unless you really need to. Also be mindful that many plug-ins (such as the "history" plugin) are actually listeners.

The actual number of jobs that can be running at any moment in time is limited by the size of the thread pool. If there are five threads in the pool, no more than five jobs can run at a time. Be careful of making a lot of threads though, as the JVM, Operating System, and CPU all have a hard time juggling lots of threads, and performance degrades as the system spends time managing threads. In most cases performance starts to tank as you get into the several hundreds of threads (or fewer if the code being executed on the threads is intensive). Be mindful that if you're running within an application server, it probably has created at least a few dozen threads of its own!

Aside from those factors, it really comes down to what your jobs DO. If your jobs take a long time to complete their work, and/or their work is very CPU-intensive, then you're obviously not going to be able to run very many jobs at once, nor very many in a given span of time.

Finally, if you just can't get enough horse-power out of one Quartz instance, you can always load-balance many Quartz instances (on separate machines). Each will run the jobs out of the shared database on a first-come first-serve basis, as quickly as the triggers need fired.

The clustering feature works best for scaling out long-running and/or cpu-intensive jobs (distributing the work-load over multiple nodes). If you need to scale out to support thousands of short-running (e.g 1 second) jobs, consider partitioning the set of jobs by using multiple distinct schedulers. Using one scheduler forces the use of a cluster-wide lock, a pattern that degrades performance as you add more clients.

So here you are this far into the answer of "how many", and I still haven't given you an actual number. And I really hate to, because of all of the variables mentioned above. So let me just say, there are installments of Quartz out there that are managing hundreds-of-thousands of Jobs and Triggers, and that are, at any given moment in time executing several dozens of jobs – without even utilizing Quartz's load-balancing capabilities. With this in mind, most people should feel confident that they can get the performance out of Quartz that they need.

I'm having issues with using Quartz via RMI

RMI can be a bit problematic, especially if you don't have an understanding of how class loading via RMI

works. I highly recommend reading all of the JavaDOC available about RMI, and strongly suggest you read the following references, dug up by a kind Quartz user (Mike Curwen)

An excellent description of RMI and codebase: http://www.kedwards.com/jini/codebase.html. One of the important points is to realize that "codebase" is used by the client!

Quick info about security managers:
http://gethelp.devx.com/techtips/java_pro/10MinuteSolutions/10min0500.asp.

And finally from the java API docs, read the docs for the RMISecurityManager
http://java.sun.com/j2se/1.3/docs/api/java/rmi/RMISecurityManager.html

The important 'take away' of the API is:

> RMI's class loader will not download any classes from remote locations if no security manager has been set.

Questions About Jobs

How can I control the instantiation of Jobs?

See org.quartz.spi.JobFactory and the org.quartz.Scheduler.setJobFactory(..) method.

How do I keep a Job from being removed after it completes?

Set the property *JobDetail.setDurability(true)* - which instructs Quartz not to delete the Job when it becomes an "orphan" (when the Job not longer has a Trigger referencing it).

How do I keep a Job from firing concurrently?

Make the job class implement StatefulJob rather than Job. Read the JavaDOC for StatefulJob for more information.

How do I stop a Job that is currently executing?

See the *org.quartz.InterruptableJob* interface, and the *Scheduler.interrupt(String, String)* method.

Questions About Triggers

How do I chain Job execution? Or, how do I create a workflow?

There currently is no "direct" or "free" way to chain triggers with Quartz. However there are several ways you can accomplish it without much effort. Below is an outline of a couple approaches:

One way is to use a listener (i.e. a TriggerListener, JobListener or SchedulerListener) that can notice the completion of a job/trigger and then immediately schedule a new trigger to fire. This approach can get a bit involved, since you'll have to inform the listener which job follows which - and you may need to worry about persistence of this information. See the listener *org.quartz.listeners.JobChainingJobListener* which ships with

Quartz - as it already has some of this functionality.

Another way is to build a Job that contains within its JobDataMap the name of the next job to fire, and as the job completes (the last step in its *execute()* method) have the job schedule the next job. Several people are doing this and have had good luck. Most have made a base (abstract) class that is a Job that knows how to get the job name and group out of the JobDataMap using pre-defined keys (constants) and contains code to schedule the identified job. This abstract Job's implementation of execute() delegates to an abstract template method such as "doWork()" (where the extending Job class's real work goes) and then it contains the code for scheduling the follow-up job. Then they simply make extensions of this class that included the work the job should do. The usage of 'durable' jobs helps the application define all the jobs and once with their proper data, without yet creating triggers to fire them (other than one trigger to fire the first job in the chain).

In the future, Quartz will provide a much cleaner way to do this, but until then, you'll have to use one of the above approaches, or think of yet another that works better for you.

Why isn't my trigger firing?

The most common reason for this is not having called *Scheduler.start()*, which tells the scheduler to start firing triggers.

The second most common reason is that the trigger or trigger group has been paused.

Daylight Saving Time and Triggers

CronTrigger and SimpleTrigger each handle daylight savings time in their own way - each in the way that is intuitive to the trigger type.

First, as a review of what daylight savings time is, please read this resource: https://secure.wikimedia.org/wikipedia/en/wiki/Daylight_saving_time_around_the_world. Some readers may be unaware that the rules are different for different nations/contents. For example, the 2005 daylight savings time started in the United States on April 3, but in Egypt on April 29. It is also important to know that not only the dates are different for different locals, but the time of the shift is different as well. Many places shift at 2:00 am, but others shift time at 1:00 am, others at 3:00 am, and still others right at midnight.

SimpleTrigger allows you to schedule jobs to fire every N milliseconds. As such, it has to do nothing in particular with respect to daylight savings time in order to "stay on schedule" - it simply keeps firing every N milliseconds. Regardless your SimpleTrigger is firing every 10 seconds, or every 15 minutes, or every hour or every 24 hours it will continue to do so. However the implication of this which confuses some users is that if your SimpleTrigger is firing say every 12 hours, before daylight savings switches it may be firing at what appears to be 3:00 am and 3:00 pm, but after daylight savings 4:00 am and 4:00 pm. This is not a bug - the trigger has kept firing exacly every N milliseconds, it just that the "name" of that time that humans impose on that moment has changed.

CronTrigger allows you to schedule jobs to fire at certain moments with respect to a "Gregorian calendar". Hence, if you create a trigger to fire every day at 10:00 am, before and after daylight savings time switches it will continue to do so. However, depending on whether it was the Spring or Autumn daylight savings event, for that particular Sunday, the actual time interval between the firing of the trigger on Sundary morning at 10:00 am since its firing on Saturday morning at 10:00 am will not be 24 hours, but will instead be 23 or 25 hours respectively.

Daylight Saving Time and Triggers

There is one additional point users must understand about CronTrigger with respect to daylight savings. This is that you should take careful thought about creating schedules that fire between midnight and 3:00 am (the critical window of time depends on your trigger's locale, as explained above). The reason is that depending on your trigger's schedule, and the particular daylight event, the trigger may be skipped or may appear to not fire for an hour or two. As examples, say you are in the United States, where daylight savings events occur at 2:00 am. If you have a CronTrrigger that fires every day at 2:15 am, then on the day of the beginning of daylight savings time the trigger will be skipped, since, 2:15 am never occurs that day. If you have a CronTrigger that fires every 15 minutes of every hour of every day, then on the day daylight savings time ends you will have an hour of time for which no triggerings occur, because when 2:00 am arrives, it will become 1:00 am again, however all of the firings during the one o'clock hour have already occurred, and the trigger's next fire time was set to 2:00 am - hence for the next hour no triggerings will occur.

In summary, all of this makes perfect sense, and should be easy to remember if you keep these two rules in mind:

- SimpleTrigger ALWAYS fires exactly every N seconds, with no relation to the time of day.
- CronTrigger ALWAYS fires at a given time of day and then computes its next time to fire. If that time does not occur on a given day, the trigger will be skipped. If the time occurs twice in a given day, it only fires once, because after firing on that time the first time, it computes the next time of day to fire on.

Other trigger types that are based on sliding along a calendar (rather than exact amounts of time), such as CalenderIntervalTrigger, will be similarly affected - but rather than missing a firing, or firing twice, may end up having it's fire time shifted by an hour.

Questions About JDBCJobStore

How do I improve the performance of JDBC-JobStore?

There are a few known ways to speed up JDBC-JobStore, only one of which is very practical.

First, the obvious, but not-so-practical:

- Buy a better (faster) network between the machine that runs Quartz, and the machine that runs your RDBMS.
- Buy a better (more powerful) machine to run your database on.
- Buy a better RDBMS.

Now for something simple, but effective: Build indexes on the Quartz tables.

Most database systems will automatically put indexes on the primary-key fields, many will also automatically do it for the foreign-key field. Make sure yours does this, or make the indexes on all key fields of every table manually.

Next, manually add some additional indexes: most important to index are the TRIGGER table's "next_fire_time" and "state" fields. Last (but not as important), add indexes to every column on the FIRED_TRIGGERS table.

```
create index idx_qrtz_t_next_fire_time on qrtz_triggers(NEXT_FIRE_TIME);
create index idx_qrtz_t_state on qrtz_triggers(TRIGGER_STATE);
create index idx_qrtz_t_nf_st on qrtz_triggers(TRIGGER_STATE,NEXT_FIRE_TIME);
```

```
create index idx_qrtz_ft_trig_name on qrtz_fired_triggers(TRIGGER_NAME);
create index idx_qrtz_ft_trig_group on qrtz_fired_triggers(TRIGGER_GROUP);
create index idx_qrtz_ft_trig_name on qrtz_fired_triggers(TRIGGER_NAME);
create index idx_qrtz_ft_trig_n_g on \
    qrtz_fired_triggers(TRIGGER_NAME,TRIGGER_GROUP);
create index idx_qrtz_ft_trig_inst_name on qrtz_fired_triggers(INSTANCE_NAME);
create index idx_qrtz_ft_job_name on qrtz_fired_triggers(JOB_NAME);
create index idx_qrtz_ft_job_group on qrtz_fired_triggers(JOB_GROUP);
create index idx_qrtz_t_next_fire_time_misfire on \
    qrtz_triggers(MISFIRE_INSTR,NEXT_FIRE_TIME);
create index idx_qrtz_t_nf_st_misfire on \
    qrtz_triggers(MISFIRE_INSTR,NEXT_FIRE_TIME,TRIGGER_STATE);
create index idx_qrtz_t_nf_st_misfire_grp on \
    qrtz_triggers(MISFIRE_INSTR,NEXT_FIRE_TIME,TRIGGER_GROUP,TRIGGER_STATE);
```

The clustering feature works best for scaling out long-running and/or cpu-intensive jobs (distributing the work-load over multiple nodes). If you need to scale out to support thousands of short-running (e.g 1 second) jobs, consider partitioning the set of jobs by using multiple distinct schedulers (and hence multiple sets of tables (with distinct prefixes)). Using one scheduler forces the use of a cluster-wide lock, a pattern that degrades performance as you add more clients.

My DB Connections don't recover properly if the database server is restarted.

If you're having Quartz create the connection data source (by specifying the connection parameters in the quartz properties file) make sure you have a connection validation query specified, such as:

```
org.quartz.dataSource.myDS.validationQuery=select 0 from dual
```

This particular query is extremely efficient for Oracle. For other databases, you'll need to think of an efficient query that always works as long as the connection is good.

If you're datasource is managed by your application server, make sure the datasource is configured in such a way that it can detect failed connections.

Questions About Transactions

I'm using JobStoreCMT and I'm seeing deadlocks, what can I do?

JobStoreCMT is in heavy use, under heavy load by many people. It is believed to be free of bugs that can cause deadlock. However, every now and then we get complaints about deadlocks. In all cases thus far, the problem has turned out to be "user error", thus the list below is some things for you to check if you are experiencing deadlocks.

- Some databases falsely detect deadlocks when a tx takes a long time. Make sure you have put indexes on your tables (see improving performance of JDBCJobStore).
- Make sure you have at least number-of-threads-in-thread-pool + 2 connections in your datasources.
- Make sure you have both a managed and non-managed datasource configured for Quartz to use.
- Make sure that all work you do with the Scheduler interface is done from within a transaction. Accomplish this by using the Scheduler within a SessionBean that has its tx settings "Required" and "Container". Or within a MessageDrivenBean with similar settings. Finally, start a UserTransaction yourself, and commit the work when done.

- If your Jobs' execute() methods use the Scheduler, make sure a transaction is in progress by using a UserTransaction or by setting the Quartz config property "org.quartz.scheduler.wrapJobExecutionInUserTransaction=true".

I'm using Oracle RAC and I'm seeing deadlocks, what can I do?

Oracle RAC has many limitations relating to transactions and locking. Quartz is known to work fine with Oracle RAC if you're careful about the setup. The QTZ-149 issue contains some discussion and links that may help you if you're having issues.

Questions about Clustering, (Scaling and High-Availability) Features

What clustering capabilities exist with Quartz?

Quartz ships with well-proven clustering capabilities that offer scaling and high availability features. You can read about these features in the Quartz Configuration Reference.

Additional clustering features that do not rely upon a backing database are available (free of cost) from Terracotta.

Questions About Spring

I'm using Quartz via Spring's scheduler wrappers, and I need help...

Check with the Spring community...

I'm seeing triggers stuck in the ACQUIRED state, or other weird data problems.

Spring defaults the Quartz property "org.quartz.jobStore.dontSetAutoCommitFalse" to "true" - which means Quartz will not turn off auto-commit mode on the database connections that it uses. This is the opposite of Quartz's own default for this setting. If your connection is defaulting to have auto-commit on, then you'll run into all sorts of strange problems relating to data inconsistencies -- the most common symptom being triggers that are "stuck" in the "ACQUIRED" state. Fix this by explicitly setting the property to "false".