Quartz Scheduler Developer Guide

Version 2.2.1

This document applies to Quartz Scheduler Version 2.2.1 and to all subsequent releases.

Specifications contained herein are subject to change and these changes will be reported in subsequent release notes or new editions.

Document ID: QS-DG-221-20141015

Table of Contents

1 Using the Quartz API

Instantiating the Scheduler

Before you can use the Scheduler, it needs to be instantiated. To do this, you use a SchedulerFactory.

Some users of Quartz keep an instance of a factory in a JNDI store, others find it just as easy (or easier) to instantiate and use a factory instance directly (as in the example below).

Once a scheduler is instantiated, it can be started, placed in stand-by mode, and shutdown. Be aware that once you shut a scheduler down, you cannot restart it without reinstantiating it. Triggers will not fire (and therefore, jobs will not execute) until the scheduler has been started. Nor will they fire while the scheduler is in the paused state.

The following code snippet 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);
```

Key Interfaces

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 want the Scheduler to execute.
- 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 its 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 "Instantiating the Scheduler" on page 6.

Quartz provides "builder" classes that define a Domain Specific Language (or DSL, also sometimes referred to as a "fluent interface"). Here is an example:

```
// 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. As shown below, this interface has one simple method:

```
package org.quartz;
public interface Job {
  public void execute(JobExecutionContext context)
    throws JobExecutionException;
}
```

When a job's trigger fires, 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, including 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. This object 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. The JobDetail object is essentially the definition of the job instance.

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 want to have.

Triggers may also have a JobDataMap associated with them. A JobDataMap is useful for passing to a job parameters 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 have both 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. Quartz was designed to create a separation between the schedule and the work to be performed on that schedule. This design has many benefits.

For example, you can create jobs and store them in the job scheduler independent of a trigger. This enables you to associate many triggers 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. This enables you to reschedule them later, without having to re-define them. 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 when 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. In other words, the complete key (or identifier) for a job or a trigger is the composed of the name and group.

For detailed information about jobs and triggers, see "Jobs and JobDetails" on page 10 and "Working with Triggers" on page 15.

2 Working with Jobs and JobDetails

Jobs and JobDetails

Jobs are 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 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.*;
```

The following code snippet defines a job and schedules it for execution:

```
// 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 shown below:

```
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.

To provide properties/configuration for a Job instance or keep track of a job's state between executions, you use 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 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 is an 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: " + myFloatValue);
    }
  }
```

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. 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 previous example
      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: " + myFloatValue);
    }
  }
```

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 previous example
      state.add(new Date());
      System.err.println("Instance " + key + " of DumbJob says: "
         + jobSays + ", and val is: " + myFloatValue);
    }
    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

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.

Each stored JobDetail is referred to as a *job definition* or *JobDetail instance*, and each executing job is a *job instance* or *instance of a job definition*. In general, when the term “job” is used, it refers to a named definition, or JobDetail. The class implementing the job interface, is called the “job class.”

Job State and Concurrency

The following are additional notes about a job's state data (aka JobDataMap) and concurrency. There are a couple annotations that you can add 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. 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 you can define 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.

3 Working with Triggers

Triggers

Like jobs, triggers are easy to work with, however they have a variety of customizable options that you need to understand before you can make full use of Quartz.

There are different types of triggers that you can select from to meet different scheduling needs. The two most common types are *simple triggers* and *cron triggers*. Details about these specific trigger types are provided in "Working with Triggers" on page 15.

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. You set these common properties 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 effect. 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 its last time on June 5th.

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

Any integer value is allowed for priority, positive or negative. If you do not set a priority on a Trigger, then it will use the default priority of 5.

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, make sure to familiarize yourself with the misfire instructions that are defined on the given trigger types, and explained in their Javadoc. Addition information about misfire instructions for specific trigger types is also provided in "Working with Triggers" on page 15.

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 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 object that implements the Calendar interface {shown below).

```
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. These 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 as shown in the example below:

```
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 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.

The org.quartz.impl.calendar package provides a number of Calendar implementations that may suit your needs.

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.*:
```

Examples of Defining Triggers with Different Schedules

The following are various examples of defining triggers with simple schedules. Review them all, as they each show at least one new/different point.

Also, 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 shown here.

Note: Note that TriggerBuilder (and Quartz's other builders) 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, TriggerBuilder will generate a random name for your trigger. Similarly, if you don't call startAt(..), then the current time (immediately) is assumed.

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), "now" is implied
  .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 future
  .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 default group
  .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);
```

SimpleTrigger Misfire Instructions

SimpleTrigger has several instructions that can be used to inform Quartz what it should do when a misfire occurs. (For information about misfires, see the About Triggers topic.) These instructions are defined as constants on SimpleTrigger itself (including Javadoc that describes their behavior). These constants include:

```
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
```

All triggers have the MISFIRE_INSTRUCTION_SMART_POLICY instruction available for use, and this instruction is the default for all trigger types.

If the 'smart policy' instruction is used, SimpleTrigger dynamically chooses between its various misfile instructions, based on the configuration and state of the given SimpleTrigger instance. The Javadoc for the SimpleTrigger.updateAfterMisfire() method explains the 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();
```

CronTriggers

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.”

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:

- Seconds
- Minutes
- Hours
- Day-of-Month
- Month
- Day-of-Week
- 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 means “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”.

Format

The fields are as follows:

Field Name	Mandatory	Allowed Values	Allowed Special Characters
Seconds	YES	0-59	, - *

Field Name	Mandatory	Allowed Values	Allowed Special Characters
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 you want your 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, you 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.

  Note: 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

Expression	Meaning
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
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 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

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_IGNORE_MISFIRE_POLICY
MISFIRE_INSTRUCTION_DO_NOTHING
MISFIRE_INSTRUCTION_FIRE_NOW
```

All triggers also have the 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();
```

4 Working with TriggerListeners and JobListeners

TriggerListeners and JobListeners

Listeners are objects that you create to perform actions based on events occurring within the scheduler. As indicated by their names, *TriggerListeners* receive events related to triggers, and *JobListeners* receive events related to jobs.

Trigger-related events include: trigger firings, trigger misfirings (discussed in "Working with Triggers" on page 15), 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);
}
```

Creating 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 at run time, and must be given a name (or rather, they must advertise their own name via their getName() method).

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

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

Note: 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.

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.

Code Samples for Adding Listeners

The following code samples illustrate ways to add JobListeners with interest in different types of jobs. Adding TriggerListeners works in a the same way.

Adding a JobListener that is interested in a particular job:

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

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", "myJobGroup")));
```

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"), jobGroupEquals("yourGroup")));
```

Adding a JobListener that is interested in all jobs

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

5 Working with SchedulerListeners

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.

Among other events, Scheduler-related events include:

- The addition of a job or trigger
- The removal of a job or trigger
- A serious error within the Scheduler
- The shutdown of the Scheduler

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

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();
}
```

Adding a SchedulerListener

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

Removing a SchedulerListener:

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

6 Working with JobStores

About Job Stores

JobStores are responsible for keeping track of all the work data you give to the scheduler: jobs, triggers, calendars, and so forth.

Selecting the appropriate JobStore for your Quartz scheduler instance is an important step. The choice is easy one once you understand the differences between them. You declare which JobStore your scheduler should use (and its configuration settings) in the properties file (or object) that you provide to the SchedulerFactory that you use to produce your scheduler instance.

Important: Never use a JobStore instance directly in your code. The JobStore is for behind-the-scenes use of Quartz itself. You have to let Quartz know (through configuration) which JobStore to use. After that, you only work with the Scheduler interface in your code.

RAMJobStore

RAMJobStore is the simplest JobStore to use. It is also offers the best performance (in terms of CPU time).

A RAMJobStore, as its name indicates, keeps all of its data in RAM. This is why it's lightning-fast and also why it is simple to configure.

The drawback to using RAMJobStore is that when your application ends (or crashes), all of the scheduling information is lost. Therefore, a 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 set the o`rg.quartz.jobStore.class` property to `org.quartz.simpl.RAMJobStore` as illustrated in the example below:

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

There are no other settings you need to worry about.

JDBCJobStore

JDBCJobStore keeps all of its data in a database via JDBC. Because it uses a database, it is a bit more complicated to configure than RAMJobStore, and it 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.

Note that in the scripts, all 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 have created the tables, 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 (for example, within a J2EE application server), you should use JobStoreCMT, which case Quartz will let the app server container manage the transactions.

Lastly, you must set up a DataSource from which JDBCJobStore can get connections to your database. DataSources are defined in your Quartz properties using one of a several 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 the application server within which Quartz is running. You do this 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 one of the following:

- `org.quartz.impl.jdbcjobstore.JobStoreTx`
- `org.quartz.impl.jdbcjobstore.JobStoreCMT`

The choice depends on the selection you made based on the explanations in the above paragraphs.

The following example shows how you configure 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. Quartz provides database-specific delegates for databases that do not operate well with StdJDBCDelegate. 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.

The following example shows how you configure 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.

The following example shows how you configure 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).

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

Tip: If your Scheduler is busy, meaning that is 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.

Tip: The org.quartz.jobStore.useProperties config parameter can be set to true (it 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 come with serializing non-String classes into a BLOB.

TerracottaJobStore

TerracottaJobStore 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. Its 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 set the `org.quartz.jobStore.class` property to `org.terracotta.quartz.TerracottaJobStore` in your Quartz configuration and add one extra property to specify the location of the Terracotta server:

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

For more information about the TerracottaJobStore, see the *Terracotta Quartz user Guide*.

7 Configuration, the Scheduler Factory, and Logging

Components to Configure

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 five 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 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). However, you want to 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 satisfactory) ThreadPool 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 very well tested. Nearly everyone using Quartz uses this pool.

JobStores and *DataSources* are discussed in "Working with JobStores" on page 35. 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, you can create 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 a server object for

RMI (make itself available to remote connections), what host and port to use, and so forth. StdSchedulerFactory (discussed below) can also produce Scheduler instances that are actually proxies (RMI stubs) to Schedulers created in remote processes. For more information about the StdSchedulerFactory, see "Scheduler Factories" on page 43.

Scheduler Factories

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 document *Quartz Scheduler Example Programs and Sample Code*.

DirectSchedulerFactory

DirectSchedulerFactory is another SchedulerFactory implementation. It is useful if you want to create your own Scheduler instance in a more programmatic way. Its use is generally discouraged for the following reasons: (1) it requires an in-depth understanding of the Scheduler and (2) it does not allow for declarative configuration, (meaning you must hard-code 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 org.quartz.plugins.history.LoggingJobHistoryPlugin and/or org.quartz.plugins.history.LoggingTriggerHistoryPlugin.

8 Miscellaneous Features of Quartz Scheduler

Plug-Ins

Quartz provides the org.quartz.spi.SchedulerPlugin interface for plugging-in additional functionality.

Plugins that ship with Quartz to provide various utility capabilities are documented in the org.quartz.plugins package.

The plugins 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 with 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 provides a number of utility jobs you can use in your application for doing things like sending e-mails and invoking EJBs.

You will find documentation for these jobs in the org.quartz.jobs package.

9 Advanced 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.

Important: Never run clustering on separate machines, unless their clocks are synchronized using some form of time-sync service or 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.

Important: 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 experience erratic behavior.

Only one node will fire the job for each firing. For example, 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 as described in "TerracottaJobStore" on page 38, and your scheduler will be 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 persistence, and running an array of Terracotta servers for HA.

The licensed edition of TerracottaJobStore provides advanced Quartz Where features, that allow for intelligent targeting of jobs to appropriate cluster nodes.

JTA Transactions

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. This applies to all jobs.

If you would like to indicate per job whether a JTA transaction should wrap its execution, then you should use the @ExecuteInJTATransaction annotation on the job class.

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.