Schwartz Plugin

Reference Documentation

GRAILS

Schwartz Plugin - Reference Documentation

Burt Beckwith

Version 1.0.1

Table of Contents

Chapter 1. Introduction to the Schwartz Plugin

The Schwartz plugin integrates the Quartz Enterprize Job Scheduler with Grails, making it easy to schedule and manage recurring and ad-hoc jobs for asynchronous and synchronous processing.

The plugin is similar at a high level to the Quartz plugin in that it makes it easy to schedule Quartz `Job`s and `Trigger`s without having to deal directly with the Quartz API and mindset. But if you are used to working with Quartz directly you can continue to do so with this plugin - it provides convenience classes and Traits to make job scheduling easier, but you have a lot of flexibility in how you perform the various tasks.

In addition to this plugin documentation, be sure to familiarize yourself with the Quartz documentation, in particular the Configuration Reference, the Cookbook, the Tutorials, the Examples, and Best Practices.

1.1. Release History

- July 12, 2016
    - 1.0.0 release

Chapter 2. Usage

Scheduling jobs in Quartz is generally done in two stages. Data about how to run your jobs (the class name and various configuration options including if the job is stateless, durable, etc.) is represented via an implementation of the JobDetail interface. But there is no scheduling information there; that's handled by classes that implement the Trigger interface. A job may have just a single `Trigger` with a configured schedule for when it fires, or it could have multiple triggers each using different schedules and possibly trigger-specific configuration. Or there might not be any triggers at all; as long as the job is durable (Quartz deletes non-durable jobs without active triggers) you can register a `JobDetail` and manually trigger the job on-demand, from other triggers, from listeners, etc.

When a trigger fires, an application class must be available to do the work, and that's handled by a class that you create in your application that implements the Job interface. Often job classes are instantiated for every invocation but this isn't a requirement, and the plugin supports using `Job` classes registered as Spring beans. If you use a singleton Spring bean be be careful not to store job-related state in the bean instance unless it won't cause problems across multiple invocations.

2.1. SchwartzJobFactory

The plugin includes an implementation of the Quartz JobFactory interface, `SchwartzJobFactory`, which manages retrieving Spring bean instances when triggers fire for jobs registered as Spring beans, and instantiating new instances of POGO/POJO classes otherwise. It will also make "job data" from multiple sources available to the job classes when they're executing.

Job data has three sources, the global values stored in the Scheduler SchedulerContext, job-specific values from a JobDataMap configured for the `JobDetail`, and trigger-specific `JobDataMap` instances configured for individual triggers. When you register the job in the scheduler you can include a `JobDataMap` with whatever information the job needs, or none if that isn't needed. Likewise when scheduling triggers you can store trigger-specific data. The three sources are merged together when a trigger fires and your job instance is created or retrieved as a bean, with global values added first, then the job data, and then the trigger data. This means that job and trigger can both override values from the `SchedulerContext`, and trigger values can override job values.

If you have setters or properties in your job class corresponding to keys in the merged data map, those values will be set for each invocation. This isn't a requirement though, and you can always access all of the merged data from the `JobExecutionContext` instance that is provided for each execution (it's the argument in the `Job` interface `void execute(JobExecutionContext context)` method that all jobs implement).

2.2. SchwartzJob

The plugin includes a trait, `SchwartzJob`, which you can implement to take advantage of its default settings and configuration which will make creating, registering and managing `JobDetail` and `Trigger` instances a lot simpler. By implementing the trait, `QuartzService` can easily create a `JobDetail` instance using the implementation class as the `jobClass`, and invoking the various getter methods defined in the trait. The initial return values have sensible default values, but you can

override any of them by creating the same method in your implementation class with custom values.

The trait also implements the InterruptableJob interface and includes an empty `interrupt()` method that you can override for any job class that should perform some actions if they are interrupted.

Quartz requires that a `Job` that is stateful indicate that using two annotations, DisallowConcurrentExecution and PersistJobDataAfterExecution. You can annotate your stateful job classes with these yourself, but the plugin includes another trait, `StatefulSchwartzJob`, which simply extends `SchwartzJob` and includes these annotations. So you can just implement `SchwartzJob` or `StatefulSchwartzJob` as needed.

The trait has one abstract method, `buildTriggers()`, which is invoked from `afterPropertiesSet()` which is called by the Spring `ApplicationContext` if your job is a Spring bean after dependencies have been injected (because the trait implements `InitializingBean`). Create your job's trigger(s) at this point and add them to the `triggers` list defined in the trait to have them auto-registered for you at startup. You can use the plugin's builder support or the Quartz builders directly to create triggers. You don't have to create any triggers though, and can just define an empty method; you can configure and schedule triggers yourself, or trigger jobs on-demand as needed.

Note that using these traits is not required - you can create (or reuse from non-Grails applications) job classes that implement the `Job` interface, and work directly with Quartz, creating and scheduling `JobDetail` and `Trigger` instances yourself. The traits and other helper classes are and are there to reduce the amount of manual configuration required. The plugin supports any implementation of the `Job` interface.

2.2.1. SchwartzJob method and property summary

Method/Property	Description
`QuartzService quartzService`	Dependency injection property for the `quartzService` bean; only valid if the job class is a Spring bean (e.g. if it's defined in `grails-app/services` or otherwise registered in Spring). For non-bean job classes the `getQuartzService()` method will retrieve the service from the `ApplicationContext` for you.
`String getJobName()`	Defines the job name for the `JobDetail` that is registered for the class; the default value is the class name without the package
`String getJobGroup()`	Defines the job group for the `JobDetail` that is registered for the class; the default is the Quartz default ("DEFAULT")
`JobKey getJobKey()`	Convenience method to create a JobKey with the values from `getJobName()` and `getJobGroup()`
`String getDescription()`	Defines the description for the `JobDetail` that is registered for the class; there is no default.
`boolean getSessionRequired()`	If `true`, a listener will be configured to start a GORM session before the job starts and flush and close it after it finishes (similar to the OpenSessionInView pattern used in controllers); defaults to `true`

Method/Property	Description
`boolean getDurable()`	If `true` the job will remain registered even if there are no active triggers associated, otherwise the `JobDetail` will be deleted by Quartz; defaults to `true`.
`boolean getRequestsRecovery()`	If `true` jobs that are executing when the scheduler stops without being properly shut down (e.g. if the app crashes) will execute again at the next startup; defaults to `false`.
`void interrupt()`	This is defined in the `InterruptableJob` interface and can be overridden to add code that runs if the job is interrupted by a call to `scheduler.interrupt()`; the default implementation is empty.
`void buildTriggers()`	Abstract in the trait and must be implemented in application job classes. Provides a hook at startup for creating triggers that should be scheduled automatically. Can be empty if you prefer to schedule triggers yourself, or if you don't need scheduled triggers and will trigger jobs on-demand.
`List<Trigger> triggers`	Add triggers here in `buildTriggers()` to make them available for automatic scheduling.
`TriggerBuilder builder()`	Creates a `TriggerBuilder` with the job name and group configured.
`TriggerBuilder builder(String triggerName)`	Creates a `TriggerBuilder` with the job name and group configured, and with the specified trigger name set.
`BuilderFactory factory()`	Creates a `BuilderFactory` with the job name and group configured.
`BuilderFactory factory(String triggerName)`	Creates a `BuilderFactory` with the job name and group configured, and with the specified trigger name set.
`JobBuilder jobBuilder()`	Creates a `JobBuilder` configured from current values from the class.
`void updateJobDetail()`	Updates the data in the Scheduler for the class using values from the getter methods.
`void updateJobDetail(JobDetail jobDetail)`	Updates the data in the Scheduler for the class using values from the `JobDetail`.
`JobDetail getStoredJobDetail()`	Retrieve the current stored job detail for the class.
`List<? extends Trigger> getStoredTriggers()`	Retrieves all triggers registered for the class.
`Date schedule(Trigger trigger)`	Adds or updates the trigger in the scheduler, returning the next fire time.
`void triggerJob()`	Executes the job immediately.
`void triggerJob(Map jobData)`	Executes the job immediately, with the specified job data available during execution.

Method/Property	Description
`QuartzService getQuartzService()`	Called implicitly for job classes that aren't Spring beans to retrieve the service from the `ApplicationContext` to ensure that the service is available in methods that use it.
`void afterPropertiesSet()`	This is defined in the `InitializingBean` interface and is called by the `ApplicationContext` at startup; calls `buildTriggers()` to let classes define triggers to be automatically scheduled.

Chapter 3. Configuration

3.1. Plugin config options

There are several configuration settings that you can set in the Grails config to customize various aspects of the plugin, and you can also specify Quartz configuration settings that will affect the configuration of the Scheduler, JobStore, etc.

All config options must be in the `quartz` block in `application.groovy` (or make the equivalent changes in `application.yml` if you haven't converted it to Groovy syntax yet) but are shown here without the prefix.

Property	Default Value	Meaning
quartz.autoStartup	`true`	If `false` the plugin will do all of its initialization tasks when the application starts (configuring Spring beans, registering `JobDetail`s and `Trigger`s, validation tasks, data purging if configured, etc.) but doesn't call `quartzScheduler.start()`. This gives you a chance to do custom work after the plugin initializes but before starting the scheduler, or disabling the scheduler per-environment (e.g. in the test environment). You can manually start later by dependency injecting the Quartz scheduler bean (using `Scheduler quartzScheduler` or `def quartzScheduler`) and calling `quartzScheduler.start()`.
quartz.clearQuartzDataOnStartup	`false`	when `true`, deletes job and trigger data from Quartz database tables (all tables except QRTZ_FIRED_TRIGGERS, QRTZ_LOCKS, and QRTZ_SCHEDULER_STATE, and filtered by scheduler name) on startup (only makes sense if jdbcStore=true)
quartz.exposeSchedulerInRepository	`false`	If `true` the `Scheduler` instance will be registered in the Quartz SchedulerRepository.
quartz.globalJobListenerNames	'exceptionPrinterJobListener', 'loggingJobListener', 'progressTrackingListener'	Names of Spring beans that implement the JobListener interface that should be registered with Quartz at startup as global listeners.

Property	Default Value	Meaning
quartz.globalTriggerListenerNames	'loggingTriggerListener', 'progressTrackingListener'	Names of Spring beans that implement the TriggerListener interface that should be registered with Quartz at startup as global listeners.
quartz.jdbcStore	`false`	If `true`, enables storing job and trigger data in a database instead of in-memory with RAMJobStore.
quartz.jdbcStoreDataSource	*none*	Spring bean name of the `DataSource` bean to use with JDBC job storage, if the bean name isn't "dataSource".
quartz.pluginEnabled	`true`	If `false` nothing is configured, for example in the test environment so jobs aren't firing during test execution.
quartz.purgeQuartzTablesOnStartup	false	More aggressive than `clearQuartzDataOnStartup`; if `true` all Quartz database tables are emptied at startup, bypassing Quartz APIs and simply running SQL queries to delete everything.
quartz.schedulerListenerNames	'loggingSchedulerListener'	Names of Spring beans that implement the SchedulerListener interface that should be registered with Quartz at startup as global listeners.
quartz.waitForJobsToCompleteOnShutdown	`false`	Whether to wait for jobs to complete when the application is shutting down.

3.2. Quartz config options

In addition to the named config options above, the plugin generates a `Properties` instance with values that you would typically otherwise store in a `quartz.properties` file. Configuring the values in this way can be much more flexible, especially if you take advantage of Groovy syntax, and also has support for environments. Set the values in the `properties` block under in the root `quartz` block; when they're read to populate the `Properties` instance they're prefixed with "org.". This listing shows a fairly complete listing of the default settings for properties that are most likely to be configured. Since this mostly includes default values it doesn't make much sense to copy this to your application's config file - it's here only as a reference.

Check out the Quartz configuration documentation for detailed information about what properties are available.

`application.groovy`

```
```

```
import org.quartz.core.QuartzSchedulerResources
import org.quartz.impl.DefaultThreadExecutor
import org.quartz.impl.jdbcjobstore.StdJDBCDelegate
import org.quartz.simpl.SimpleInstanceIdGenerator
import org.quartz.simpl.SimpleThreadPool

quartz {
   autoStartup = true
   clearQuartzDataOnStartup = false
   exposeSchedulerInRepository = false
   jdbcStore = false
   jdbcStoreDataSource = 'dataSource'
   purgeQuartzTablesOnStartup = false
   startupDelay = 0
   waitForJobsToCompleteOnShutdown = false

   properties {
      // RAMJobStore
      jobStore {
         misfireThreshold = 60000
      }
      // JDBC
      /*
      jobStore {
         acquireTriggersWithinLock = false // should be true if
                                           // batchTriggerAcquisitionMaxCount > 1
         clusterCheckinInterval = 7500
         dontSetAutoCommitFalse = false
         dontSetNonManagedTXConnectionAutoCommitFalse = false
         driverDelegateClass = StdJDBCDelegate.name
         driverDelegateInitString = null
         isClustered = false
         lockHandler.class = null // if null Quartz will determine which to use
         maxMisfiresToHandleAtATime = 20
         misfireThreshold = 60000
         selectWithLockSQL = 'SELECT * FROM {0}LOCKS WHERE SCHED_NAME = {1}' +
                             ' AND LOCK_NAME = ? FOR UPDATE'
         tablePrefix = 'QRTZ_'
         txIsolationLevelReadCommitted = false
         txIsolationLevelSerializable = false
         useProperties = false
      }
      */
      managementRESTService {
         bind = '0.0.0.0:9889'
         enabled = false
      }
      scheduler {
         batchTriggerAcquisitionFireAheadTimeWindow = 0
         batchTriggerAcquisitionMaxCount = 1
         classLoadHelper.class = null // Quartz default is CascadingClassLoadHelper
```

```
but
                                    // Spring's SchedulerFactoryBean configures a
                                    // ResourceLoaderClassLoadHelper if no value is
set
         dbFailureRetryInterval = 15000
         idleWaitTime = 30000
         instanceId = 'NON_CLUSTERED'
         instanceIdGenerator.class = SimpleInstanceIdGenerator.name
         instanceName = 'QuartzScheduler'
         interruptJobsOnShutdown = false
         interruptJobsOnShutdownWithWait = false
         jmx {
            export = true // default is false
            objectName = null // if null Quartz will generate with
                              // QuartzSchedulerResources.generateJMXObjectName()
            proxy = false
            proxy.class = null
         }
         makeSchedulerThreadDaemon = false
         rmi {
            bindName = null // if null Quartz will generate with
                            // QuartzSchedulerResources.getUniqueIdentifier()
            createRegistry = QuartzSchedulerResources.CREATE_REGISTRY_NEVER // 'never'
            export = false
            proxy = false
            registryHost = 'localhost'
            registryPort = 1099
            serverPort = -1 // random
         }
         skipUpdateCheck = true
         threadName = instanceName + '_QuartzSchedulerThread'
         threadsInheritContextClassLoaderOfInitializer = false
         userTransactionURL = null
         wrapJobExecutionInUserTransaction = false
      }
      threadExecutor.class = DefaultThreadExecutor.name
      threadPool.class = SimpleThreadPool.name
      threadPool {
         makeThreadsDaemons = false
         threadCount   = 10
         threadPriority = Thread.NORM_PRIORITY // 5
         threadsInheritContextClassLoaderOfInitializingThread = false
         threadsInheritGroupOfInitializingThread = true
      }
   }
}
```

Chapter 4. Comparison with the Quartz Plugin

The two plugins are similar in many ways, but significantly different in others. Both make it easy to create a simple class that is conveniently registered with Quartz as a `JobDetail` and optionally some triggers.

The Quartz plugin abstracts away some of the details with the expectation that this makes the process simpler. I've found the opposite however; for the small number of keystrokes saved the abstractions used add layers of dynamic Groovy code that add little value, slow down execution (admittedly not significantly), and make working with Quartz more confusing in some cases.

Using Quartz outside of Grails in a regular Java project isn't particularly complex. You just have to create a class for each job and implement the org.quartz.Job interface which has a single method, `execute` with a single argument, a `JobExecutionContext`, for example:

```
import org.quartz.Job;
import org.quartz.JobExecutionContext;
import org.quartz.JobExecutionException;

public class MyJob implements Job {
   public void execute(JobExecutionContext context) throws JobExecutionException {
      // your code here
   }
}
```

The Quartz plugin doesn't require that your job classes implement `Job` or any interface, but simply have an `execute` method. It can take a `JobExecutionContext` argument if you have a need for it, or can be a no-arg method if not, and there is no mention of `JobExecutionException` anywhere in the documentation. A job can be (and typically is) implemented as simply as

```
class MyJob {

   def execute() {
      // your code here
   }
}
```

This certainly seems better, using a pattern that's common in Grails and Groovy apps where Groovy and/or the framework reduce boilerplate code and let you focus on solving the problem at hand, in this case performing some actions when a trigger fires for your job. Internally the plugin has a single class that implements `Job` (and a similar second class for stateful jobs) which is the actual class registered in Quartz as the implementation class for the job. Your application's job classes are registered as Spring beans, and the bean name is stored with the `JobDetail`, and when a trigger fires, the `execute()` method is invoked in the plugin's job class, and your application's job bean is retrieved from Spring to do the actual job processing work. The `execute()` method and whether it

has a `JobExecutionContext` argument are looked up once using reflection at startup, and then invoked with reflection each time a trigger fires, with handling for `InvocationTargetException` and `IllegalAccessException` that's necessary because the method is dynamically invoked.

There some other hidden costs also. In addition to the `execute()` method, you define triggers for your job using the plugin's custom DSL in a static Closure:

```
class MyJob {

   static triggers = {
      simple name:'simpleTrigger', startDelay: ...
   }

   def execute() {
      // your code here
   }
}
```

Having a closure like that in the class means you cannot use the `CompileStatic` annotation at the class level, only per-method. This is a small difference and there wouldn't be much gained if the trigger definition code was statically compiled, but in practice there tends to be little use of `CompileStatic` at all. I now use `CompileStatic` wherever I can, because of the performance boost but also because it makes writing code much faster because IDEs have a lot more information available for use with autocomplete and other inferencing features. My opinion is that plugins should be encouraging and enabling *more* use of static compilation, not implicitly discouraging it.

Hopefully your jobs will always run without errors, but exceptions occur and by ignoring the possibility that a `JobExecutionException` might be thrown during job execution also has a runtime cost. This is a checked exception, which Groovy doesn't require us to deal with. But it implements that by wrapping checked exceptions in an `UndeclaredThrowableException`, and unwrapping them before they're actually handled. This also affects the majority of the methods in `grails.plugins.quartz.QuartzJob` and `grails.plugins.quartz.JobManagerService`, specifically the methods that call Quartz `Scheduler` methods, most of which can throw `JobExecutionException`. Exceptions in Groovy are already slow and expensive and having a feature in a plugin that adds processing and overhead only makes the problem worse.

None of these costs are particularly bad, and in general performance is likely not noticeably different from the more verbose implementation in Java. But if a lot of work is being done in your Quartz jobs, or if they fire often, these costs can certainly add up. This could easily be shrugged off as a non-issue if the implementation savings was significant, but all we gained was the ability to omit the `execute()` method argument, omit the `implements Job` declaration, and omit the `JobExecutionException`. That's a negligible savings (literally a few dozen keystrokes at best), especially given the potential costs.

4.1. Building triggers

Triggers are defined statically in the Quartz plugin in a `triggers` closure in each job class, using the plugin's trigger DSL. Only "simple" and "cron" triggers are directly supported, but you can

configure any trigger type by specifying the class and all of the trigger properties as name/value pairs. The DSL is fairly limited and there is no autocompletion.

With Schwartz you also can define triggers in the job class, typically in the `buildTriggers()` method in job classes that implement `SchwartzJob` or `StatefulSchwartzJob`. There is no DSL, but instead you define triggers programmatically. As you can see from the examples throughout the documentation, using the Quartz builder classes added in Quartz 2.0, or the plugin's `BuilderFactory` which aggregates the logic for all of the Quartz builders into one builder, the programmatic approach feels a lot like using a DSL but with the added benefit of having immediate feedback about available options and syntax when coding in an IDE. You can create different triggers or use different settings depending on the current `Environment`, get access to the `Config`, and use any Spring bean, Grails service, etc. as needed, and the `QuartzService` is available inside all `SchwartzJob` classes. You can also create instances of the "Impl" classes directly and set properties there, for example if you already have working existing code, but this approach shouldn't be used in general.

You can additionally create triggers outside of your job classes and schedule them on-demand, using the Quartz `Scheduler` or helper methods in `QuartzService`.

4.2. Differences in default values

When building triggers, if no value is specified the Quartz plugin defaults to a 60,000 millisecond (one minute) repeat interval, "GRAILS_JOBS" for the job group name, "GRAILS_TRIGGERS" for the trigger group name, "Grails Job" for the job description, and, inexplicably, "0 0 6 * * ?" for the CRON expression. This plugin has no defaults for these properties (other than the job and trigger names, which default to the Quartz default value of "DEFAULT" for both).

Another difference is trigger priority; triggers created in `grails.plugins.quartz.TriggerUtils` default to a priority value of 6 (one larger than the default value of 5, presumably as an homage to Spinal Tap).

The two plugins have the same default values for start delay (0), repeat count (repeat forever), session required (`true`), durability (`true`), and recovery requested (`false`).

4.3. Migrating from the Quartz plugin

4.3.1. Migrating config settings

The Quartz plugin's config settings (`quartz.autoStartup`, `quartz.exposeSchedulerInRepository`, `quartz.jdbcStore`, `quartz.jdbcStoreDataSource`, `quartz.pluginEnabled`, `quartz.purgeQuartzTablesOnStartup`, and `quartz.waitForJobsToCompleteOnShutdown`) are all also used in this plugin and have the same default values, so there shouldn't be any changes required there.

4.3.2. Migrating job classes

Quartz plugin jobs are not directly usable in the Schwartz plugin but are easily converted. The Quartz plugin includes an artifact handler that lets you (technically it requires that you) define job classes under `grails-app/jobs`. Schwartz jobs are not artifacts by default and have no dedicated class location.

The first thing to do for each job is to decide if it should be a Grails Service or not. This is optional but if your job writes to the database it should be transactional or use transactional services to do that work, and if it makes sense to do that work transactionally inside the job class then a Service is a good option.

Run the create-job script for each existing Quartz plugin job, and add the `--pogo` flag to generate the class under `src/main/groovy`, or omit the flag to generate a Service; if your Quartz plugin job is a stateful job (somewhat confusingly referred to as a non-concurrent job in the Quartz plugin docs, since concurrency is only one aspect of a stateful job) then include the `--stateful` flag, or omit it to make the generated class a stateless job:

A stateless job that's also a Service:

```
$ grails create-job <classname>
```

A stateless job defined in src/main/groovy:

```
$ grails create-job <classname> --pogo
```

A stateful job that's also a Service:

```
$ grails create-job <classname> --stateful
```

A stateful job defined in src/main/groovy:

```
$ grails create-job <classname> --pogo --stateful
```

Both plugins have the same defaults for durability, session required, requests recovery, so if you had overridden any of those, do the same in your `SchwarzJob` class by overridding the corresponding method from the trait, e.g.

```
boolean getSessionRequired() { false }
```

Also add a method override for the description, job name, and job group as needed.

4.3.3. Migrating triggers

Convert triggers defined using the Quartz plugin DSL with plugins triggers defined using this plugin's `BuilderFactory` (either directly or by using one of the `factory()` methods defined in `SchwartzJob`) or using the Quartz builders (either directly or by using one of the `builder()` methods defined in `SchwartzJob`). Define these in the `buildTriggers()` method and add the trigger instances to the `triggers` list, e.g.

```
import static com.agileorbit.schwartz.builder.MisfireHandling.NowWithExistingCount
import static org.quartz.DateBuilder.todayAt

void buildTriggers() {
   triggers << factory('cron every second').cronSchedule('0/1 * * * * ?').build()

   triggers <<
factory('Repeat3TimesEvery100').intervalInMillis(100).repeatCount(3).build()

   triggers << factory('repeat every 500ms forever').intervalInMillis(500).build()

   triggers << factory('repeat every two days forever').intervalInDays(2).build()

   triggers << factory('trigger1')
         .intervalInMillis(100)
         .startDelay(2000).noRepeat()
         .jobData(foo: 'bar').build()

   triggers << factory('run_once_immediately').noRepeat().build()

   triggers << factory('MisfireTrigger2')
         .intervalInMillis(150)
         .misfireHandling(NowWithExistingCount)
         .build()

   triggers << factory('trigger1').group('group1').intervalInSeconds(1).build()

   triggers << factory('run every day one second before midnight')
         .startAt(todayAt(23,59,59))
         .intervalInDays(1).build()

   triggers << factory('run every day at 11:00 AM')
         .startAt( todayAt(11,00,00).before(new Date()) ? tomorrowAt(11,00,00) :
todayAt(11,00,00) )
         .intervalInDays(1).build()
}
```

Chapter 5. Triggers

Quartz includes four interfaces that extend the core Trigger interface; CalendarIntervalTrigger, CronTrigger, DailyTimeIntervalTrigger, and SimpleTrigger. Each has a default implementation (CalendarIntervalTriggerImpl, CronTriggerImpl, DailyTimeIntervalTriggerImpl, and SimpleTriggerImpl) and you can directly instantiate them and configure their properties yourself. They all include a few parameterized constructors but these constructors and this approach in general are deprecated in favor of using their builder classes.

The builders support a fluent API with method chaining by returning `this` from mutator methods, and you end up with readable and intuitive code to define triggers, and IDE autocompletion also greatly helps with quickly configuring settings. These include the 'core' builder TriggerBuilder and a scheduler builder that handles the schedule-related configuration for each different trigger type (CalendarIntervalScheduleBuilder, CronScheduleBuilder, DailyTimeIntervalScheduleBuilder, and SimpleScheduleBuilder). DateBuilder is also available and has a large number of methods that help with working with dates so you don't have to work with the frustrating Java date apis. Using these builders also tends to help avoid making invalid combinations of settings, but that's still possible of course.

5.1. Using builders to create triggers

The general flow for creating any of the four types of triggers is to start with a call to the static `TriggerBuilder.newTrigger()` method to create the initial `TriggerBuilder` instance, and then call the various instance methods to configure job detail properties and common schedule-related properties, e.g.

```
TriggerBuilder.newTrigger()
      .forJob('some job name')
      .withPriority(3)
```

Using static imports can help a lot to compact your builder code, e.g.

```
import static org.quartz.TriggerBuilder.newTrigger

newTrigger()
      .forJob('some job name')
      .withPriority(3)
```

The instance methods are designed to be chained and all mutator methods return `this` instead of being `void`; this isn't required but is the preferred way to work with the builders, so all of the examples will use this approach. At some point in the method call chain you can join in a schedule builder with a call to `withSchedule()`, e.g.

```
import static org.quartz.CronScheduleBuilder.dailyAtHourAndMinute
import static org.quartz.TriggerBuilder.newTrigger

newTrigger()
      .forJob('some job name')
      .withPriority(3)
      .withSchedule(dailyAtHourAndMinute(2, 30))
```

The schedule builders have one or more static methods that are used to create an instance with all of the initial properties required for the particular schedule pattern. In this example I used a method that will implicitly generate a CRON expression using the three properties I provided (daily repeat, repeat hour, and repeat minute) but I could have used a different initializing method for weekly repeating, monthly, etc., or the simplest option, providing a CRON expression directly:

```
import static org.quartz.CronScheduleBuilder.cronSchedule
import static org.quartz.TriggerBuilder.newTrigger

newTrigger()
      .forJob('some job name')
      .withPriority(3)
      .withSchedule(cronSchedule('0 15 10 * * ?'))
      .build()
```

Once everything is configured, call `build()` to instantiate and configure the trigger type associated with the schedule builder you used:

```
import org.quartz.CronTrigger

import static org.quartz.CronScheduleBuilder.dailyAtHourAndMinute
import static org.quartz.TriggerBuilder.newTrigger

CronTrigger cronTrigger = newTrigger()
      .forJob('some job name')
      .withPriority(3)
      .withSchedule(dailyAtHourAndMinute(2, 30))
      .build()
```

The builders use generics, so in the example above no cast is needed even if your code is annotated with `@CompileStatic`.

The following sections describe the various initial static methods and subsequent instance methods for each of the builders

5.2. CalendarIntervalScheduleBuilder

5.2.1. Method Summary

Listing 1. CalendarIntervalScheduleBuilder static methods

```
static CalendarIntervalScheduleBuilder calendarIntervalSchedule()
```

Listing 2. CalendarIntervalScheduleBuilder instance methods

```
// interval, IntervalUnit

CalendarIntervalScheduleBuilder withInterval(int interval, IntervalUnit unit)
CalendarIntervalScheduleBuilder withIntervalInDays(int intervalInDays)
CalendarIntervalScheduleBuilder withIntervalInHours(int intervalInHours)
CalendarIntervalScheduleBuilder withIntervalInMinutes(int intervalInMinutes)
CalendarIntervalScheduleBuilder withIntervalInMonths(int intervalInMonths)
CalendarIntervalScheduleBuilder withIntervalInSeconds(int intervalInSeconds)
CalendarIntervalScheduleBuilder withIntervalInWeeks(int intervalInWeeks)
CalendarIntervalScheduleBuilder withIntervalInYears(int intervalInYears)

// misfireInstruction

CalendarIntervalScheduleBuilder withMisfireHandlingInstructionIgnoreMisfires()
CalendarIntervalScheduleBuilder withMisfireHandlingInstructionDoNothing()
CalendarIntervalScheduleBuilder withMisfireHandlingInstructionFireAndProceed()

// TimeZone

CalendarIntervalScheduleBuilder inTimeZone(TimeZone timezone)

// preserveHourOfDayAcrossDaylightSavings

CalendarIntervalScheduleBuilder preserveHourOfDayAcrossDaylightSavings(boolean
preserveHourOfDay)

// skipDayIfHourDoesNotExist
CalendarIntervalScheduleBuilder skipDayIfHourDoesNotExist(boolean skipDay)
```

Table 1. Corresponding BuilderFactory methods/properties

Builder method	BuilderFactory methods/properties
`static calendarIntervalSchedule`	N/A, this is always called because it's the only static builder method
`withInterval`	`interval`, `unit`
`withIntervalInMinutes`	`intervalInMinutes`
`withIntervalInHours`	`intervalInHours`
`withIntervalInDays`	`intervalInDays`
`withIntervalInWeeks`	`intervalInWeeks`

Builder method	BuilderFactory methods/properties
`withIntervalInMonths`	`intervalInMonths`
`withIntervalInYears`	`intervalInYears`
`withMisfireHandlingInstructionIgnoreMisfires`	`misfireHandling(IgnoreMisfires)`
`withMisfireHandlingInstructionDoNothing`	`misfireHandling(DoNothing)`
`withMisfireHandlingInstructionFireAndProceed`	`misfireHandling(FireAndProceed)`
`preserveHourOfDayAcrossDaylightSavings`	`preserveHourOfDay()`
`skipDayIfHourDoesNotExist`	`skipDay`

Use these static imports when using these `BuilderFactory` methods:

```
grails.plugin.schwartz.builder.MisfireHandling.*
org.quartz.DateBuilder.IntervalUnit.*
```

5.3. CronScheduleBuilder

5.3.1. Method Summary

Listing 3. CronScheduleBuilder static methods

```
static CronScheduleBuilder atHourAndMinuteOnGivenDaysOfWeek(int hour, int minute,
Integer... daysOfWeek)
static CronScheduleBuilder cronSchedule(CronExpression cronExpression)
static CronScheduleBuilder cronSchedule(String cronExpression)
static CronScheduleBuilder cronScheduleNonvalidatedExpression(String cronExpression)
static CronScheduleBuilder dailyAtHourAndMinute(int hour, int minute)
static CronScheduleBuilder monthlyOnDayAndHourAndMinute(int dayOfMonth, int hour, int
minute)
static CronScheduleBuilder weeklyOnDayAndHourAndMinute(int dayOfWeek, int hour, int
minute)
```

Listing 4. CronScheduleBuilder instance methods

```
// timeZone

CronScheduleBuilder inTimeZone(TimeZone timezone)

// misfireInstruction

CronScheduleBuilder withMisfireHandlingInstructionIgnoreMisfires()
CronScheduleBuilder withMisfireHandlingInstructionDoNothing()
CronScheduleBuilder withMisfireHandlingInstructionFireAndProceed()
```

Table 2. Corresponding BuilderFactory methods/properties

Builder method	BuilderFactory methods/properties
`static cronSchedule(String)`	`cronSchedule`
`static cronScheduleNonvalidatedExpression`	`cronScheduleNonvalidated`
`static cronSchedule(CronExpression)`	`cronExpression`
`static dailyAtHourAndMinute`	`hourAndMinuteMode(DailyAt), hour, minute`
`static atHourAndMinuteOnGivenDaysOfWeek`	`hourAndMinuteMode(DaysOfWeek), hour, minute, days`
`static weeklyOnDayAndHourAndMinute`	`hourAndMinuteMode(Weekly), day, hour, minute`
`static monthlyOnDayAndHourAndMinute`	`hourAndMinuteMode(Monthly), day, hour, minute`
`inTimeZone`	`timeZone`
`withMisfireHandlingInstructionIgnoreMisfires`	`misfireHandling(IgnoreMisfires)`
`withMisfireHandlingInstructionDoNothing`	`misfireHandling(DoNothing)`
`withMisfireHandlingInstructionFireAndProceed`	`misfireHandling(FireAndProceed)`

Use these static imports when using these `BuilderFactory` methods:

```
grails.plugin.schwartz.builder.HourAndMinuteMode.*
grails.plugin.schwartz.builder.MisfireHandling.*
org.quartz.DateBuilder.SUNDAY
org.quartz.DateBuilder.MONDAY
...
org.quartz.DateBuilder.SATURDAY
```

5.4. DailyTimeIntervalScheduleBuilder

5.4.1. Method Summary

Listing 5. DailyTimeIntervalScheduleBuilder static methods

```
static DailyTimeIntervalScheduleBuilder dailyTimeIntervalSchedule()
```

Listing 6. DailyTimeIntervalScheduleBuilder instance methods

```
// interval

DailyTimeIntervalScheduleBuilder withInterval(int interval, IntervalUnit unit)
DailyTimeIntervalScheduleBuilder withIntervalInHours(int intervalInHours)
DailyTimeIntervalScheduleBuilder withIntervalInMinutes(int intervalInMinutes)
DailyTimeIntervalScheduleBuilder withIntervalInSeconds(int intervalInSeconds)

// IntervalUnit

DailyTimeIntervalScheduleBuilder withInterval(int interval, IntervalUnit unit)

// daysOfWeek

DailyTimeIntervalScheduleBuilder onDaysOfTheWeek(Integer ... onDaysOfWeek)
DailyTimeIntervalScheduleBuilder onDaysOfTheWeek(Set<Integer> onDaysOfWeek)
DailyTimeIntervalScheduleBuilder onEveryDay()
DailyTimeIntervalScheduleBuilder onMondayThroughFriday()
DailyTimeIntervalScheduleBuilder onSaturdayAndSunday()

// startTimeOfDay

DailyTimeIntervalScheduleBuilder startingDailyAt(TimeOfDay timeOfDay)

// endTimeOfDay

DailyTimeIntervalScheduleBuilder endingDailyAfterCount(int count)
DailyTimeIntervalScheduleBuilder endingDailyAt(TimeOfDay timeOfDay)

// repeatCount

DailyTimeIntervalScheduleBuilder withRepeatCount(int repeatCount)

// misfireInstruction

DailyTimeIntervalScheduleBuilder withMisfireHandlingInstructionDoNothing()
DailyTimeIntervalScheduleBuilder withMisfireHandlingInstructionFireAndProceed()
DailyTimeIntervalScheduleBuilder withMisfireHandlingInstructionIgnoreMisfires()
```

Table 3. Corresponding BuilderFactory methods/properties

Builder method	BuilderFactory methods/properties
`static dailyTimeIntervalSchedule`	N/A, this is always called because it's the only static builder method
`withInterval`	`interval, unit`
`withIntervalInSeconds`	`intervalInSeconds`
`withIntervalInMinutes`	`intervalInMinutes`

Builder method	BuilderFactory methods/properties
`withIntervalInHours`	`intervalInHours`
`onDaysOfTheWeek(Set<Integer>)`	not used; Collections and single ints are converted to `Integer[]`
`onDaysOfTheWeek(Integer …)`	`days`
`onMondayThroughFriday`	`mondayThroughFriday()`
`onSaturdayAndSunday`	`saturdayAndSunday()`
`onEveryDay`	`everyDay()`
`startingDailyAt`	`dailyStart`
`endingDailyAt`	`dailyEnd`
`endingDailyAfterCount`	`dailyEndAfterCount`
`withMisfireHandlingInstructionIgnoreMisfires`	`misfireHandling(IgnoreMisfires)`
`withMisfireHandlingInstructionDoNothing`	`misfireHandling(DoNothing)`
`withMisfireHandlingInstructionFireAndProceed`	`misfireHandling(FireAndProceed)`
`withRepeatCount`	`repeatCount`

Use these static imports when using these `BuilderFactory` methods:

```
grails.plugin.schwartz.builder.MisfireHandling.*
```

5.5. SimpleScheduleBuilder

5.5.1. Method Summary

Listing 7. SimpleScheduleBuilder static methods

```
static SimpleScheduleBuilder repeatHourlyForever()
static SimpleScheduleBuilder repeatHourlyForever(int hours)
static SimpleScheduleBuilder repeatHourlyForTotalCount(int count)
static SimpleScheduleBuilder repeatHourlyForTotalCount(int count, int hours)
static SimpleScheduleBuilder repeatMinutelyForever()
static SimpleScheduleBuilder repeatMinutelyForever(int minutes)
static SimpleScheduleBuilder repeatMinutelyForTotalCount(int count)
static SimpleScheduleBuilder repeatMinutelyForTotalCount(int count, int minutes)
static SimpleScheduleBuilder repeatSecondlyForever()
static SimpleScheduleBuilder repeatSecondlyForever(int seconds)
static SimpleScheduleBuilder repeatSecondlyForTotalCount(int count)
static SimpleScheduleBuilder repeatSecondlyForTotalCount(int count, int seconds)
static SimpleScheduleBuilder simpleSchedule()
```

Listing 8. SimpleScheduleBuilder instance methods

```
// interval

SimpleScheduleBuilder withIntervalInHours(int intervalInHours)
SimpleScheduleBuilder withIntervalInMilliseconds(long intervalInMillis)
SimpleScheduleBuilder withIntervalInMinutes(int intervalInMinutes)
SimpleScheduleBuilder withIntervalInSeconds(int intervalInSeconds)

// repeatCount

SimpleScheduleBuilder repeatForever()
SimpleScheduleBuilder withRepeatCount(int triggerRepeatCount)

// misfireInstruction

SimpleScheduleBuilder withMisfireHandlingInstructionFireNow()
SimpleScheduleBuilder withMisfireHandlingInstructionIgnoreMisfires()
SimpleScheduleBuilder withMisfireHandlingInstructionNextWithExistingCount()
SimpleScheduleBuilder withMisfireHandlingInstructionNextWithRemainingCount()
SimpleScheduleBuilder withMisfireHandlingInstructionNowWithExistingCount()
SimpleScheduleBuilder withMisfireHandlingInstructionNowWithRemainingCount()
```

Table 4. Corresponding BuilderFactory methods/properties

Builder method	BuilderFactory methods/properties
`static simpleSchedule`	`called if RepeatMode isn't specified`
`static repeatMinutelyForever`	`repeatMode(Minutes), repeatForever()`
`static repeatMinutelyForever(minutes)`	`repeatMode(Minutes), repeatForever(), minutes`
`static repeatSecondlyForever`	`repeatMode(Seconds), repeatForever()`
`static repeatSecondlyForever(seconds)`	`repeatMode(Seconds), repeatForever(), seconds`
`static repeatHourlyForever`	`repeatMode(Hours), repeatForever()`
`static repeatHourlyForever(hours)`	`repeatMode(Hours), repeatForever(), hours`
`static repeatMinutelyForTotalCount(count)`	`repeatMode(Minutes), totalCount`
`static repeatMinutelyForTotalCount(count, minutes)`	`repeatMode(Minutes), totalCount, minutes`
`static repeatSecondlyForTotalCount(count)`	`repeatMode(Seconds), totalCount`
`static repeatSecondlyForTotalCount(count, seconds)`	`repeatMode(Seconds), totalCount, seconds`
`static repeatHourlyForTotalCount(count)`	`repeatMode(Hours), totalCount`
`static repeatHourlyForTotalCount(count, hours)`	`repeatMode(Hours), totalCount, hours`
`withIntervalInMilliseconds`	`intervalInMillis`
`withIntervalInSeconds`	`intervalInSeconds`
`withIntervalInMinutes`	`intervalInMinutes`

Builder method	BuilderFactory methods/properties
withIntervalInHours	intervalInHours
withRepeatCount	repeatCount
repeatForever	repeatForever() (or omit since it's the default)
withMisfireHandlingInstructionIgnoreMisfires	simpleMisfireHandling(IgnoreMisfires)
withMisfireHandlingInstructionFireNow	simpleMisfireHandling(FireNow)
withMisfireHandlingInstructionNextWithExistingCount	simpleMisfireHandling(NextWithExistingCount)
withMisfireHandlingInstructionNextWithRemainingCount	simpleMisfireHandling(NextWithRemainingCount)
withMisfireHandlingInstructionNowWithExistingCount	simpleMisfireHandling(NowWithExistingCount)
withMisfireHandlingInstructionNowWithRemainingCount	simpleMisfireHandling(NowWithRemainingCount)

Use these static imports when using these `BuilderFactory` methods:

```
grails.plugin.schwartz.builder.SimpleMisfireHandling.*
grails.plugin.schwartz.builder.RepeatMode.*
```

5.6. TriggerBuilder

5.6.1. Method Summary

Listing 9. TriggerBuilder static methods

```
static TriggerBuilder<Trigger> newTrigger()
```

Listing 10. TriggerBuilder instance methods

```
// TriggerKey

TriggerBuilder<T> withIdentity(String name)
TriggerBuilder<T> withIdentity(String name, String group)
TriggerBuilder<T> withIdentity(TriggerKey triggerKey)

// description

TriggerBuilder<T> withDescription(String triggerDescription)

// priority

TriggerBuilder<T> withPriority(int triggerPriority)

// calendarName

TriggerBuilder<T> modifiedByCalendar(String calName)

// startTime

TriggerBuilder<T> startAt(Date triggerStartTime)
TriggerBuilder<T> startNow()

// endTime

TriggerBuilder<T> endAt(Date triggerEndTime)

// scheduleBuilder

<SBT extends T> TriggerBuilder<SBT> withSchedule(ScheduleBuilder<SBT> schedBuilder)

// JobKey

TriggerBuilder<T> forJob(JobDetail jobDetail)
TriggerBuilder<T> forJob(JobKey keyOfJobToFire)
TriggerBuilder<T> forJob(String jobName)
TriggerBuilder<T> forJob(String jobName, String jobGroup)

// JobDataMap

TriggerBuilder<T> usingJobData(JobDataMap newJobDataMap)
TriggerBuilder<T> usingJobData(String dataKey, Boolean value)
TriggerBuilder<T> usingJobData(String dataKey, Double value)
TriggerBuilder<T> usingJobData(String dataKey, Float value)
TriggerBuilder<T> usingJobData(String dataKey, Integer value)
TriggerBuilder<T> usingJobData(String dataKey, Long value)
TriggerBuilder<T> usingJobData(String dataKey, String value)
```

Table 5. Corresponding BuilderFactory methods/properties

Builder method	BuilderFactory methods/properties
`static newTrigger`	N/A, this is always called because it's the only static builder method
`withIdentity(name)`	`name`
`withIdentity(name, group)`	`name, group`
`withIdentity(TriggerKey)`	`triggerKey`
`withDescription`	`description`
`withPriority`	`priority`
`modifiedByCalendar`	`calendarName`
`startAt`	`startAt`
`startNow`	`startNow()` (or omit since it's the default)
`endAt`	`endAt`
`withSchedule`	`N/A`
`forJob(JobKey)`	`jobKey`
`forJob(jobName)`	`jobName`
`forJob(jobName, jobGroup)`	`jobName, jobGroup`
`forJob(JobDetail)`	`jobDetail`
`usingJobData(String, String)`	`none, use jobData Map`
`usingJobData(String, Integer)`	`none, use jobData Map`
`usingJobData(String, Long)`	`none, use jobData Map`
`usingJobData(String, Float)`	`none, use jobData Map`
`usingJobData(String, Double)`	`none, use jobData Map`
`usingJobData(String, Boolean)`	`none, use jobData Map`
`usingJobData(JobDataMap)`	`jobDataMap`

5.7. DateBuilder

5.7.1. Method Summary

Listing 11. DateBuilder static methods

```
static DateBuilder newDate()
static DateBuilder newDateInTimezone(TimeZone tz)
static DateBuilder newDateInLocale(Locale lc)
static DateBuilder newDateInTimeZoneAndLocale(TimeZone tz, Locale lc)
```

Listing 12. DateBuilder instance methods

```
// TimeZone

DateBuilder inTimeZone(TimeZone timezone)

// Locale

DateBuilder inLocale(Locale locale)

// month

DateBuilder inMonth(int inMonth)
DateBuilder inMonthOnDay(int inMonth, int onDay)

// day

DateBuilder onDay(int onDay)
DateBuilder inMonthOnDay(int inMonth, int onDay)

// year

DateBuilder inYear(int inYear)

// hour

DateBuilder atHourOfDay(int atHour)
DateBuilder atHourMinuteAndSecond(int atHour, int atMinute, int atSecond)

// minute

DateBuilder atMinute(int atMinute)
DateBuilder atHourMinuteAndSecond(int atHour, int atMinute, int atSecond)

// second

DateBuilder atSecond(int atSecond)
DateBuilder atHourMinuteAndSecond(int atHour, int atMinute, int atSecond)
```

Listing 13. DateBuilder static utility methods

```
static DateBuilder newDate()
static DateBuilder newDateInTimezone(TimeZone tz)
static DateBuilder newDateInLocale(Locale lc)
static DateBuilder newDateInTimeZoneAndLocale(TimeZone tz, Locale lc)

static Date dateOf(int hour, int minute, int second)
static Date dateOf(int hour, int minute, int second, int dayOfMonth, int month)
static Date dateOf(int hour, int minute, int second, int dayOfMonth, int month, int
year)
static Date evenHourDate(Date date)
static Date evenHourDateAfterNow()
static Date evenHourDateBefore(Date date)
static Date evenMinuteDate(Date date)
static Date evenMinuteDateAfterNow()
static Date evenMinuteDateBefore(Date date)
static Date evenSecondDate(Date date)
static Date evenSecondDateAfterNow()
static Date evenSecondDateBefore(Date date)
static Date futureDate(int interval, IntervalUnit unit)
static Date nextGivenMinuteDate(Date date, int minuteBase)
static Date nextGivenSecondDate(Date date, int secondBase)
static Date todayAt(int hour, int minute, int second)
static Date tomorrowAt(int hour, int minute, int second)
static Date translateTime(Date date, TimeZone src, TimeZone dest)
```

5.8. BuilderFactory

Quartz added trigger builder classes in 2.0 that have chainable methods (the mutator methods return `this`) and provide a fluent and intuitive approach to creating triggers. In practice I found several methods to be a bit verbose, and thought it would be better in many cases to combine all of the builders so there would be a single starting point for all types of triggers. The plugin's `BuilderFactory` (an awkward name, but certainly better than `BuilderBuilder` ...) provides this.

If you're building triggers inside a `SchwartzJob` class the best way to initialize a `BuilderFactory` is with one of the two `builder()` methods since they set the job name and group, and optionally the trigger name for you:

```
Trigger trigger = builder('mytrigger').some().builder().methods().build()
```

You can also start with just the constructor and do everything explicitly, e.g.

```
Trigger trigger = new BuilderFactory().some().builder().methods().build()
```

All of the options in the various triggers are configurable as properties or chainable methods, so the standard Groovy Map constructor is an option also, either explicitly followed by a call to the `build()`

instance method, or in one step using the static `build(Map)` method, e.g.

```
Trigger trigger = BuilderFactory.build(
      name: triggerName, group: triggerGroup, jobName: jobName, jobGroup: jobGroup,
      startAt: start, endAt: end, cronSchedule: cron, timeZone: timeZone,
      description: description, jobDataMap: jobDataMap, calendarName: calendarName,
      misfireHandling: IgnoreMisfires, priority: priority)
```

which is equivalent to

```
Trigger trigger = builder()
   .name(triggerName)
   .group(triggerGroup)
   .jobName(jobName)
   .jobGroup(jobGroup)
   .startAt(start)
   .endAt(end)
   .cronSchedule(cron)
   .timeZone(timeZone)
   .description(description)
   .jobDataMap(jobDataMap)
   .calendarName(calendarName)
   .priority(priority)
   .misfireHandling(IgnoreMisfires)
   .build()
```

Because all trigger properties and methods are available when using `BuilderFactory`, it is possible to inadvertently set properties or call methods that apply to two or more trigger types; this is invalid and when you call the `build()` method, the code that validates the configured properties and determines which type of trigger to build will detect the problem and throw an exception with detailed information about what values were applicable for the different trigger types to help you to fix the mistakes.

The following tables summarize the various methods and corresponding chainable methods that are available for the different trigger types, and the common non-schedule-related methods applicable to all trigger types.

Table 6. CalendarIntervalTrigger schedule properties and methods

Property	Method
`Integer interval`	`BuilderFactory interval(int _)`
`Integer intervalInDays`	`BuilderFactory intervalInDays(int _)`
`Integer intervalInHours`	`BuilderFactory intervalInHours(int _)`
`Integer intervalInMinutes`	`BuilderFactory intervalInMinutes(int _)`
`Integer intervalInMonths`	`BuilderFactory intervalInMonths(int _)`
`Integer intervalInSeconds`	`BuilderFactory intervalInSeconds(int _)`

Property	Method
`Integer intervalInWeeks`	`BuilderFactory intervalInWeeks(int _)`
`Integer intervalInYears`	`BuilderFactory intervalInYears(int _)`
`MisfireHandling misfireHandling`	`BuilderFactory misfireHandling(MisfireHandling _)`
`Boolean preserveHour`	`BuilderFactory preserveHour(boolean _ = true)`
`Boolean skipDay`	`BuilderFactory skipDay(boolean _ = true)`
`TimeZone timeZone`	`BuilderFactory timeZone(TimeZone _)`
`IntervalUnit unit`	`BuilderFactory unit(IntervalUnit _)`

Table 7. CronTrigger schedule properties and methods

Property	Method
`CronExpression cronExpression`	`BuilderFactory cronExpression(CronExpression _)`
`String cronSchedule`	`BuilderFactory cronSchedule(String _)`
`String cronScheduleNonvalidated`	`BuilderFactory cronScheduleNonvalidated(String _)`
`Integer day`	`BuilderFactory day(int _)`
`def days`	`BuilderFactory days(_)` (also `void setDays(daysOfWeek)` which accepts a single int, an `Integer[]` array, or `Collection` and converts to `Integer[]`, and `Integer[] getDays()`)
`Integer hour`	`BuilderFactory hour(int _)`
`HourAndMinuteMode hourAndMinuteMode`	`BuilderFactory hourAndMinuteMode(HourAndMinuteMode _)`
`Integer minute`	`BuilderFactory minute(int _)`
`MisfireHandling misfireHandling`	`BuilderFactory misfireHandling(MisfireHandling _)`
`TimeZone timeZone`	`BuilderFactory timeZone(TimeZone _)`

Table 8. DailyTimeIntervalTrigger schedule properties and methods

Property	Method
`TimeOfDay dailyEnd`	`BuilderFactory dailyEnd(TimeOfDay _)`
`Integer dailyEndAfterCount`	`BuilderFactory dailyEndAfterCount(int _)`
`TimeOfDay dailyStart`	`BuilderFactory dailyStart(TimeOfDay _)`
`def days`	`BuilderFactory days(_)` (also `void setDays(daysOfWeek)` which accepts a single int, an `Integer[]` array, or `Collection` and converts to `Integer[]`, and `Integer[] getDays()`)
`Boolean everyDay`	`BuilderFactory everyDay(boolean _ = true)`
`Integer interval`	`BuilderFactory interval(int _)`
`Integer intervalInHours`	`BuilderFactory intervalInHours(int _)`
`Integer intervalInMinutes`	`BuilderFactory intervalInMinutes(int _)`
`Integer intervalInSeconds`	`BuilderFactory intervalInSeconds(int _)`

Property	Method
`MisfireHandling misfireHandling`	`BuilderFactory misfireHandling(MisfireHandling _)`
`Boolean mondayThroughFriday`	`BuilderFactory mondayThroughFriday(boolean _ = true)`
`Integer repeatCount`	`BuilderFactory repeatCount(int _)`
`Boolean saturdayAndSunday`	`BuilderFactory saturdayAndSunday(boolean _ = true)`
`IntervalUnit unit`	`BuilderFactory unit(IntervalUnit _)`

Table 9. SimpleTrigger properties and methods

Property	Method
`Integer hours`	`BuilderFactory hours(int _)`
`Integer intervalInHours`	`BuilderFactory intervalInHours(int _)`
`Long intervalInMillis`	`BuilderFactory intervalInMillis(long _)`
`Integer intervalInMinutes`	`BuilderFactory intervalInMinutes(int _)`
`Integer intervalInSeconds`	`BuilderFactory intervalInSeconds(int _)`
`Integer minutes`	`BuilderFactory minutes(int _)`
`MisfireHandling misfireHandling`	`BuilderFactory misfireHandling(MisfireHandling _)`
`Integer repeatCount`	`BuilderFactory repeatCount(int _)`
`Boolean repeatForever`	`BuilderFactory repeatForever(boolean _ = true)`
`RepeatMode repeatMode`	`BuilderFactory repeatMode(RepeatMode _)`
`Integer seconds`	`BuilderFactory seconds(int _)`
`Integer totalCount`	`BuilderFactory totalCount(int _)`

Table 10. Trigger properties and methods

Property	Method
`String calendarName`	`BuilderFactory calendarName(String _)`
`String description`	`BuilderFactory description(String _)`
`Date endAt`	`BuilderFactory endAt(Date _)`
`String group`	`BuilderFactory group(String _)`
`SchwartzJob job`	`BuilderFactory job(SchwartzJob _)`
`Map<String, ?> jobData`	`BuilderFactory jobData(Map<String, ?> _)`
`JobDataMap jobDataMap`	`BuilderFactory jobDataMap(JobDataMap _)`
`JobDetail jobDetail`	`BuilderFactory jobDetail(JobDetail _)`
`String jobGroup`	`BuilderFactory jobGroup(String _)`
`JobKey jobKey`	`BuilderFactory jobKey(JobKey _)`
`String jobName`	`BuilderFactory jobName(String _)`
`TriggerKey key`	`BuilderFactory key(TriggerKey _)`
`String name`	`BuilderFactory name(String _)`

Property	Method
`Integer priority`	`BuilderFactory priority(int _)`
`Date startAt`	`BuilderFactory startAt(Date _)`

Table 11. Utility methods

Method	Description
`BuilderFactory startDelay(int millis)`	Chainable builder method that sets the start time of the trigger as the current time plus the specified milliseconds
`void setStartDelay(int millis)`	Traditional setter method that sets the start time of the trigger as the current time plus the specified milliseconds
`BuilderFactory startNow()`	Essentially a no-op since the default start date is `new Date()`.
`BuilderFactory noRepeat()`	Resets the repeat count to zero (the default is to repeat forever)

Chapter 6. Listeners

The plugin includes a few implementations of the Quartz listener interfaces and support for others. The listeners are all enabled by default but can be disabled (and additional listeners can be enabled) in the Configuration.

6.1. com.agileorbit.schwartz.listener.LoggingJobListener

Implements JobListener and logs current state information at the debug level for each method.

6.2. com.agileorbit.schwartz.listener.LoggingTriggerListener

Implements TriggerListener and logs current state information at the debug level for each method.

6.3. com.agileorbit.schwartz.listener.LoggingSchedulerListener

Implements SchedulerListener and logs method parameter data at the debug level for each method.

6.4. com.agileorbit.schwartz.listener.ExceptionPrinterJobListener

Implements JobListener and logs the `JobKey` and the exception stacktrace if an exception occurs while a job is executing.

6.5. com.agileorbit.schwartz.listener.SessionBinderJobListener

Implements JobListener and works like the web-tier open-session-in-view pattern, binding a GORM session before your job starts and closing it when it finishes. Will be active for each `SchwartzJob` only if the job class returns `true` from its `getSessionRequired()` method (defaults to `true`) defined in the trait.

6.6. com.agileorbit.schwartz.listener.QuartzListeners

Aggregate interface that implements `JobListener`, `TriggerListener`, and `SchedulerListener`. `com.agileorbit.schwartz.listener.QuartzListenersAdaptor` is an adaptor class that implements `QuartzListeners` and delegates each method to the corresponding method in an instance of JobListenerSupport, SchedulerListenerSupport, or TriggerListenerSupport.

Chapter 7. QuartzService

The plugin includes a utility service, `quartzService`, that it uses internally for much of the interaction with the Quartz `Scheduler` and other tasks, and you can also use it in your application. Add a dependency injection for the service like for any Spring bean (`QuartzService quartzService` or `def quartzService`). The service's methods are described below.

Method	Description
`init`	Called from `doWithApplicationContext()` at startup; should not be called from application code.
`validateTables`	Checks that the required tables exist if JDBC storage is enabled.
`validateExistingJobs`	Checks that all JobDetails can be retrieved to catch problems early such as deleted/moved/renamed Job classes.
`scheduleJobs`	Registers a `JobDetail` and schedules triggers for each Spring bean that implements `SchwartzJob`; should not be called from application code.
`scheduleJob`	Builds and registers a `JobDetail` with values from a `SchwartzJob` instance and and schedules the job's triggers.
`updateJobDetail`	Replaces data for an existing `JobDetail` with new values.
`getStoredJobDetail`	Retrieves the `JobDetail` stored for a job if one exists.
`triggerJob`	Triggers a job to run immediately.
`getTriggers`	Retrieve the triggers scheduled for a a job.
`scheduleTrigger`	Register a `Trigger` in the `Scheduler`.
`purgeTables`	Deletes data from all database tables; called at startup if `purgeQuartzTablesOnStartup` is `true` but can be called on-demand also.
`clearData`	Uses standard Quartz functionality to clear job and trigger data (all tables except QRTZ_FIRED_TRIGGERS, QRTZ_LOCKS, and QRTZ_SCHEDULER_STATE).
`tableNamePrefix`	Looks up the database table name prefix if set in the config and returns the default ("QRTZ_") if not overridden.

Chapter 8. Tutorials

8.1. Basic Tutorial

8.1.1. Create your Grails application.

```
$ grails create-app schwartztest
$ cd schwartztest
```

8.1.2. “Install” the plugin by adding it to build.gradle

Add a dependency for the plugin by adding it to the `dependencies` block in build.gradle:

```
buildscript {
   repositories {
      ...
   }
   dependencies {
      classpath "org.grails:grails-gradle-plugin:$grailsVersion"
      ...
      classpath 'com.agileorbit:schwartz:1.0.1'
   }
}

dependencies {
   ...
   compile 'com.agileorbit:schwartz:1.0.1'
   ...
}
```

8.1.3. Create a job that’s a transactional service

Next we’ll create some job classes. First, we’ll create a job that’s a transactional service:

```
$ grails create-job Skroob
```

Note that even though Schwartz jobs aren’t Grails artifacts and there is no enforced naming convention, the create-job script adds the “Job” suffix for you if the specified name doesn’t end in “Job”, and “JobService” when creating jobs in `grails-app/services`, but this is not a requirement (other than the general Grails requirement that service names end in “Service”), just a convenience.

The script will create the class under `grails-app/services` in the specified package, or in the default package if none is specified (as in this case), with commented-out example triggers that aren’t shown here:

```
package schwartztest

import com.agileorbit.schwartz.SchwartzJob
import grails.transaction.Transactional
import groovy.transform.CompileStatic
import groovy.util.logging.Slf4j
import org.quartz.JobExecutionContext
import org.quartz.JobExecutionException

@CompileStatic
@Slf4j
class SkroobJobService implements SchwartzJob {

   @Transactional
   void execute(JobExecutionContext context) throws JobExecutionException {
   }

   void buildTriggers() {
   }
}
```

Edit the file to add a simple trigger that fires every second, and a println in the `execute` method to verify that it's working:

```
package schwartztest

import com.agileorbit.schwartz.SchwartzJob
import grails.transaction.Transactional
import groovy.transform.CompileStatic
import groovy.util.logging.Slf4j
import org.quartz.JobExecutionContext
import org.quartz.JobExecutionException

@CompileStatic
@Slf4j
class SkroobJobService implements SchwartzJob {

   @Transactional
   void execute(JobExecutionContext context) throws JobExecutionException {
      println "$context.trigger.key/$context.jobDetail.key at ${new Date()}"
   }

   void buildTriggers() {
      triggers << factory('Skroob_EverySecond').intervalInSeconds(1).build()
   }
}
```

Run the application to verify that the job is registered and that it fires every second.

8.1.4. Create a stateful job that's a POGO

This is really two steps in one since a job can be stateful or stateless, and a service or defined in `src/main/groovy` as a POGO, but we'll create one job to reduce the overall length of this tutorial.

Run this to create the job:

```
$ grails create-job DarkHelmet --pogo --stateful
```

which will be generated in `src/main/groovy/schwartztest/DarkHelmetJob.groovy` (shown here with comments removed):

```
package schwartztest

import com.agileorbit.schwartz.StatefulSchwartzJob
import groovy.transform.CompileStatic
import groovy.util.logging.Slf4j
import org.quartz.JobExecutionContext
import org.quartz.JobExecutionException
import org.springframework.stereotype.Component

@CompileStatic
@Component
@Slf4j
class DarkHelmetJob implements StatefulSchwartzJob {

   void execute(JobExecutionContext context) throws JobExecutionException {
   }

   void buildTriggers() {
   }
}
```

As before, add a trigger in `buildTriggers()` and a println statement in `execute()`

```
package schwartztest

import com.agileorbit.schwartz.StatefulSchwartzJob
import groovy.transform.CompileStatic
import groovy.util.logging.Slf4j
import org.quartz.JobExecutionContext
import org.quartz.JobExecutionException
import org.springframework.stereotype.Component

@CompileStatic
@Component
@Slf4j
class DarkHelmetJob implements StatefulSchwartzJob {

   void execute(JobExecutionContext context) throws JobExecutionException {
      println "$context.trigger.key/$context.jobDetail.key at ${new Date()}"
   }

   void buildTriggers() {
      triggers << factory('DarkHelmet_Every2Second').intervalInSeconds(2).build()
   }
}
```

Because this is not a service, it won't be automatically discovered by the plugin and won't be registered in Quartz, but there are a few options. Note that the class is annotated with the Spring Component annotation; this is not required and can be removed, but if you've enabled component scanning, e.g. by annotating your `Application` class:

```
package schwartztest

import grails.boot.GrailsApp
import grails.boot.config.GrailsAutoConfiguration
import groovy.transform.CompileStatic
import org.springframework.context.annotation.ComponentScan

@CompileStatic
@ComponentScan('schwartztest')
class Application extends GrailsAutoConfiguration {
   static void main(String[] args) {
      GrailsApp.run this, args
   }
}
```

then your job class will be registered as a Spring bean and because it implements `SchwartzJob` it will be auto-registered in Quartz at startup.

If you prefer, you can register the job as a Spring bean in `grails-app/conf/spring/resources.groovy`, e.g.

```
import schwartztest.DarkHelmetJob

beans = {
   darkHelmetJob(DarkHelmetJob) {
      quartzService = ref('quartzService')
   }
}
```

8.1.5. Create a non-bean job

There's no requirement that jobs be registered as Spring beans, it's just a convenience. You can manage job registration and scheduling entireley yourself if you prefer. Run the `create-job` script again to create a POGO job class:

```
$ grails create-job Barf --pogo
```

and add a println statement in `execute()` but don't create any triggers, and delete the `Component` annotation:

```
package schwartztest

import com.agileorbit.schwartz.SchwartzJob
import groovy.transform.CompileStatic
import groovy.util.logging.Slf4j
import org.quartz.JobExecutionContext
import org.quartz.JobExecutionException

@CompileStatic
@Slf4j
class BarfJob implements SchwartzJob {

   void execute(JobExecutionContext context) throws JobExecutionException {
      println "$context.trigger.key/$context.jobDetail.key at ${new Date()}"
   }

   void buildTriggers() {
   }
}
```

Now you have several options. You could manually register the job yourself, e.g. in `BootStrap` (or later at runtime, e.g. in a service):

```
import com.agileorbit.schwartz.QuartzService
import org.quartz.SchedulerException
import schwartztest.BarfJob

class BootStrap {

   QuartzService quartzService

   def init = {
      BarfJob job = new BarfJob()
      try {
         quartzService.scheduleJob job
      }
      catch (SchedulerException e) {
         log.error e.message, e
      }
   }
}
```

or you could create a `JobDetail` and optionally some triggers and register the job using those (and you could do the same for other job classes that implement `Job` but not `SchwartzJob`, even Java classes):

```
import com.agileorbit.schwartz.QuartzService
import org.quartz.JobDetail
import org.quartz.SchedulerException
import org.quartz.Trigger
import schwartztest.BarfJob

class BootStrap {

   QuartzService quartzService

   def init = {

      BarfJob job = new BarfJob()

      JobDetail jobDetail = job.jobBuilder().build()
      Collection<? extends Trigger> triggers = ...

      try {
         quartzService.scheduleJob jobDetail, triggers, false
      }
      catch (SchedulerException e) {
         log.error e.message, e
      }
   }
}
```

Once the job is registered, you can trigger the job to run immediately whenever you want:

```
import com.agileorbit.schwartz.QuartzService
import static org.quartz.JobKey.jobKey

class SomeService {

   QuartzService quartzService

   void someMethod() {
      quartzService.triggerJob jobKey('BarfJob')
   }
}
```

8.2. JDBC Job Storage Tutorial

There's not much to do to change from in-memory job storage to database storage, but be sure to read the JDBC Job Storage section of the docs before starting this tutorial.

In this example MySQL is used but you can use any of the database types supported by Quartz. Very few details are database-specific, primarily just the initial database creation steps.

If you don't already have a MySQL database to work with, run these commands in a MySQL client as the root user or another user with permission to create databases and grant permissions:

```
> create database quartz_jdbc_test;

> grant all on quartz_jdbc_test.* to quartztest@localhost identified by 'quartztest';
```

8.2.1. Configure the DataSource

Add a dependency in `build.gradle` for the JDBC driver for your database (be sure to use the latest version available):

```
dependencies {
   ...
   runtime 'mysql:mysql-connector-java:5.1.39'
   ...
}
```

and update `application.groovy` (or make the equivalent changes in `application.yml` if you haven't converted it to Groovy syntax yet) to use the correct driver class, dialect, url, username, and password:

```
dataSource {
   ...
   dialect = org.hibernate.dialect.MySQL5InnoDBDialect
   driverClassName = 'com.mysql.jdbc.Driver'
   password = 'quartztest'
   url = 'jdbc:mysql://localhost/quartz_jdbc_test'
   username = 'quartztest'
   ...
}
```

8.2.2. Create the database tables

You'll need to create the database tables that Quartz uses to store job and trigger information. As described in the JDBC Job Storage section there are three options, using one of the Quartz scripts from the full distribution, or using one of the plugin commands. Use whichever approach you prefer and apply the changelog or execute the SQL to create the tables.

If you're using the database-migration plugin (or Liquibase directly), you should use the create-jdbc-tables-changelog command, e.g.

```
$ grails create-jdbc-tables-changelog grails-app/migrations/quartz_jdbc.groovy
```

Add the file to your main changelog file and run

```
$ grails dbm-update
```

The create-jdbc-sql command is similar but only has a dependency on Hibernate. Run the script to generate the SQL:

```
$ grails create-jdbc-sql quartz_jdbc.sql
```

and use the MySQL client or another tool to run the SQL statements and create the tables.

8.2.3. Enable database storage

Once the database tables are available, set the `jdbcStore` config property to `true`:

```
quartz {
   jdbcStore = true
}
```

Since MySQL uses the default `DriverDelegate`, it's not required to specify the delegate class name, but if you're using a database that has a custom implementation (e.g. PostgreSQL) then you would also set that, e.g.

```
import org.quartz.impl.jdbcjobstore.PostgreSQLDelegate

quartz {
   jdbcStore = true
   properties {
      jobStore {
         driverDelegateClass = PostgreSQLDelegate.name
      }
   }
}
```

8.3. Cluster Tutorial

Be sure to read the Clustering section of the docs before starting this tutorial.

8.3.1. Configure database job storage

Database job storage is required before configuring a Quartz cluster, so configure that first using the JDBC Job Storage Tutorial (or by making the required changes in an existing application).

8.3.2. Enable clustering

Next, enable clustering by setting `isClustered` to `true` in the config, and set the `instanceName` property to define the name for the cluster. For now, also set the value for `instanceId` to "AUTO" to let Quartz assign a unique value for each server node:

```
quartz {
   jdbcStore = true
   properties {
      jobStore {
         isClustered = true
      }
      scheduler {
         instanceId = 'AUTO'
         instanceName = 'ClusterTutorial'
      }
   }
}
```

8.3.3. Start the app

Enable logging for the Quartz classes in `logback.groovy`:

```
logger 'org.quartz', INFO
```

and start the app and you should see output similar to this:

```
INFO org.quartz.core.QuartzScheduler - Quartz Scheduler v.2.2.3 created.
INFO org.quartz.core.QuartzScheduler - Scheduler meta-data: Quartz Scheduler (v2.2.3)
'ClusterTutorial' with instanceId '...'
  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 10 threads.
  Using job-store 'org.springframework.scheduling.quartz.LocalDataSourceJobStore' -
which supports persistence. and is clustered.
INFO org.quartz.core.QuartzScheduler - Scheduler ClusterTutorial_$_... started.
```

8.3.4. Create a simple job

Stop the app and create a simple job that we can use to verify that clustering is enabled and working:

```
$ grails create-job ClusterTest
```

```
package clustertest

import com.agileorbit.schwartz.SchwartzJob
import grails.transaction.Transactional
import groovy.transform.CompileStatic
import groovy.util.logging.Slf4j
import org.quartz.JobExecutionContext
import org.quartz.JobExecutionException

@CompileStatic
@Slf4j
class ClusterTestJobService implements SchwartzJob {

   @Transactional
   void execute(JobExecutionContext context) throws JobExecutionException {
      println "$context.trigger.key/$context.jobDetail.key at ${new Date()}"
   }

   void buildTriggers() {
      triggers << factory('ClusterTest_EverySecond').intervalInSeconds(1).build()
   }
}
```

8.3.5. Run two instance to check that clustering is configured

Start the app again and verify that the job fires and prints to the console every second:

```
$ grails run-app --port=8081
```

Open a second terminal and run a second instance of the app on another port and verify that the job runs in both instances, but that each time the trigger fires it only executes on one of the running instances:

```
$ grails run-app --port=8082
```

Quartz makes no guarantees about load distribution among cluster nodes and in general will tend to run jobs on the first instance they happen to run on, so you may not see any output in the second window. But if you stop the first server instance (either with a clean shutdown or more forcibly) then you will definitely see that the job starts firing on the second instance, possibly after a delay of a few seconds.

8.3.6. Change from automatic instance ids to explicitly set values

Edit `application.groovy` and remove the `instanceId` setting:

```
quartz {
   jdbcStore = true
   properties {
      jobStore {
         isClustered = true
      }
      scheduler {
         instanceName = 'ClusterTutorial'
      }
   }
}
```

and add this block to `build.gradle` to enable passing system properties from the commandline:

```
configure(bootRun) {
   systemProperties System.properties
}
```

Start the two instances again, but this time passing the cluster node id in addition to the server port:

```
$ grails -Dquartz.properties.scheduler.instanceId=node1 run-app --port=8081
```

and

```
$ grails -Dquartz.properties.scheduler.instanceId=node2 run-app --port=8082
```

and you should see from the logging output that clustering is enabled and that the specified node names are used instead of randomly assigned values, and that the work of running the sample job is split between the two instances.

Chapter 9. Advanced Topics

9.1. JDBC Job Storage

The default implementation of the Quartz JobStore is RAMJobStore which keeps everything in-memory at the server where the schedular is running. This works well when you only have one server but when you need to scale beyond one, or run multiple instances for high availability, it's important to both spread the workload between the servers and to ensure that triggers only fire on one machine, and for that you need centralized storage. Using Quartz's support for storage in a relational database is one good option, and Quartz also supports using Terracotta as the `JobStore`.

The initial configuration to change from in-memory to a database is only a few steps, and there are many optional tuning and configuration options described in the documentation.

One step when using traditional Quartz outside of Grails is deciding what transaction support to use; `JobStoreTX` manages transactions for Quartz table updates but cannot participate in your application's transactions, or `JobStoreCMT` which uses the transaction support from the application server or framework and can update Quartz tables transactionally along with application table updates. You also would add connection and pool-management properties to the Quartz configuration to create a DataSource that Quartz uses for its queries.

Spring provides a third option with its LocalDataSourceJobStore implementation which uses the application's DataSource (or one of them if there are multiple) and existing transaction support, which lets Quartz updates use the same transaction support as the rest of the application. This is the option used by the plugin, and to enable it you simply set the DataSource as a property of the SchedulerFactoryBean Spring bean. The plugin will do this for you if you set the `quartz.jdbcStore` config value to `true`:

```
quartz {
   jdbcStore = true
}
```

By default it will use the `dataSource` Spring bean, and to use a different DataSource bean, set its name in the `quartz.jdbcStoreDataSource` property:

```
quartz {
   jdbcStore = true
   jdbcStoreDataSource = 'someOtherDataSource'
}
```

If you use this approach, do not configure the `org.quartz.jobStore.class` Quartz property or any values for a connection pool as described in the Quartz JDBC documentation - the plugin and the Spring support classes make the appropriate config changes when creating the Scheduler and its support objects.

9.1.1. Database tables

You'll need to create the database tables that Quartz uses to store job and trigger information, and there are a few different options available. The full Quartz distribution (available to download from the Quartz website) contains files with SQL statements for many popular databases. In addition, the plugin includes two commands that will create a Liquibase changelog or the equivalent SQL for you. Use whichever of these approaches you prefer and apply the changelog or execute the SQL to create the tables.

If you're using the database-migration plugin (or Liquibase directly), you should use the create-jdbc-tables-changelog command, e.g.

```
$ grails create-jdbc-tables-changelog grails-app/migrations/quartz_jdbc.groovy
```

Add the file to your main changelog file and run

```
$ grails dbm-update
```

The create-jdbc-sql command is similar but only has a dependency on Hibernate. Run the script to generate the SQL:

```
$ grails create-jdbc-sql quartz_jdbc.sql
```

and use the client for you database or another tool to run the SQL statements and create the tables.

9.1.2. DriverDelegate

Quartz needs to know which DriverDelegate implementation that the `JobStore` will use to perform the actual queries and updates. This is similar to the `Dialect` concept in Hibernate, and allows the core functionality to focus on the logic around job management and delegates database-specific JDBC work to a specialized helper class. Some databases use the default implementation, StdJDBCDelegate, but there are custom implementations for databases that require different logic for various calls. The following table summarizes which class to use for several popular databases but check with the online documentation to be sure to select the correct type for the database you're using:

DriverDelegate Class	Database(s)
StdJDBCDelegate	H2, Informix, MySQL
DB2v6Delegate	DB2 6.x
DB2v7Delegate	DB2 7.x
DB2v8Delegate	DB2 8.x
MSSQLDelegate	SQL Server

DriverDelegate Class	Database(s)
OracleDelegate	Oracle
PostgreSQLDelegate	PostgreSQL

The default is `StdJDBCDelegate` so if your database uses that you don't need to make any changes. If you're using one of the databases that has a custom driver delegate implementation, specify the class name in the config in the `driverDelegateClass` setting, e.g. for PostgreSQL you would use

```
import org.quartz.impl.jdbcjobstore.PostgreSQLDelegate

quartz {
   jdbcStore = true
   properties {
      jobStore {
         driverDelegateClass = PostgreSQLDelegate.name
      }
   }
}
```

There are several other JDBC-related config options, so familiarize yourself with those in case they can be of use, but the defaults should work well in general for most of the properties. The properties all start with "org.quartz.jobStore.", e.g. "org.quartz.jobStore.tablePrefix", "org.quartz.jobStore.txIsolationLevelSerializable", etc. so they would be set in the "quartz.properties.jobStore" config block, e.g.

```
quartz {
   ...
   properties {
      jobStore {
         acquireTriggersWithinLock = ...
         clusterCheckinInterval = ...
         ...
      }
   }
}
```

To summarize, the general steps to change job storage from in-memory to JDBC are:

- execute the SQL statements (from one of the Quartz example files or using one of the plugin commands) to create the tables and indexes
- set `quartz.jdbcStore` to `true` in the config
- set `quartz.driverDelegateClass` to the name of the `DriverDelegate` used by your database if it's not the default
- optionally specify a non-default DataSource bean name for `quartz.jdbcStoreDataSource`
- optionally configure additional properties

9.2. Clustering

Clustering is a moderately complex topic and you should read the Quartz clustering documentation before trying to implement it, but there are only a few required steps to get basic clustering working.

The first is to configure storage of job data in a database, so refer to the previous section for those steps.

In addition, you must enable clustering with the `isClustered` config setting, and choose a shared cluster name by setting the `instanceName` config setting, e.g.

```
quartz {
   jdbcStore = true
   properties {
      jobStore {
         isClustered = true
      }
      scheduler {
         instanceName = 'cluster_tutorial'
      }
   }
}
```

The remaining required config setting is a unique instance id for each node in the cluster, using the `instanceId` config setting. You can let Quartz assign a unique name for each node automatically by using the value “AUTO”, e.g.

```
quartz {
   jdbcStore = true
   properties {
      jobStore {
         isClustered = true
      }
      scheduler {
         instanceName = 'cluster_tutorial'
         instanceId = 'AUTO'
      }
   }
}
```

but if you want to choose the values yourself you would omit that setting from `application.groovy` and set the value externally for each server instance.

That’s all that’s required, but refer to the Quartz docs for information about the various optional config settings that are available, and try out the Cluster Tutorial to see clustering in action.

To summarize, the general steps to change job storage from in-memory to JDBC are:

- configure database job storage
- set `quartz.properties.jobStore.isClustered` to `true` in the config
- set `quartz.properties.scheduler.instanceName` to the cluster name in the config
- set `quartz.properties.scheduler.instanceId` to 'AUTO' in the config to use auto-generated unique node names, or specify the value externally for each node
- optionally configure additional properties

Chapter 10. Scripts and Commands

The plugin must be in the Gradle classpath in order for the scripts and commands to be available from the commandline. Be sure to include a "classpath" element in the `dependencies` section of the `buildscript` block in your application's `build.gradle` to make the script available to Gradle, in addition to adding a regular "compile" element in the main `dependencies` block:

```
buildscript {
   repositories {
      ...
   }
   dependencies {
      classpath "org.grails:grails-gradle-plugin:$grailsVersion"
      ...
      classpath 'com.agileorbit:schwartz:1.0.1'
   }
}

dependencies {
   ...
   compile 'com.agileorbit:schwartz:1.0.1'
   ...
}
```

10.1. create-job

Purpose

Creates a new Quartz Job class. The general format is:

```
$ grails create-job JOB_CLASS_NAME [--pogo] [--stateful]
```

Examples

```
$ grails create-job report
```

```
$ grails create-job com.mycompany.foo.priceUpdate --pogo
```

```
$ grails create-job com.mycompany.foo.priceUpdate --stateful
```

```
$ grails create-job com.mycompany.foo.priceUpdate --pogo --stateful
```

Description

Creates a class with the specified class name and package that implements either `SchwartzJob` or `StatefulSchwartzJob`. By default it will create a stateless class but will make it stateful if you include the `--pogo` flag.

By default the class will be generated as a Grails Service under `grails-app/services`, but if you include the `--pogo` flag the class will be generated under `src/main/groovy`. The plugin doesn't automatically register your job classes as Spring beans, but often job classes need to access the database or call services and other Spring beans. By making your class a Grails Service you get support for dependency injection and can easily make your job class transactional with the `Transactional` annotation. All beans that implement one of the traits will be auto-discovered at startup by `QuartzService` and a `JobDetails` and any configured `Trigger` instances will be registered in the Quartz `Scheduler` if they are not already.

10.2. create-jdbc-tables-changelog

Purpose

Creates a Liquibase changelog (or the SQL generated from a changelog) to create Quartz database tables. The general format is:

```
$ grails create-jdbc-tables-changelog [FILENAME]
```

Examples

```
$ grails create-jdbc-tables-changelog grails-app/migrations/quartz_jdbc.groovy
```

```
$ grails create-jdbc-tables-changelog grails-app/migrations/quartz_jdbc.xml
```

```
$ grails create-jdbc-tables-changelog quartz_jdbc.sql
```

Description

Creates in-memory Liquibase model objects and builds a changelog from those in Groovy or XML format (or any format supported by Liquibase). Will optionally generate the resulting SQL from the changelog. The format is determined by the extension of the file you specify; if it is "groovy", "xml", or any of the file formats supported by Liquibase, a changelog file will be created. Unlike the database-migration plugin the file is not written relative to the `grails-app/migrations` folder, it's written relative to the application root directory. If the extension of the file is "sql", a changelog will be generated but it will be used to by Liquibase to generate the SQL that would be created for the changelog and that SQL is written to the output file.

The liquibase-core jar or the database-migration plugin must be in the classpath to use this command.

10.3. create-jdbc-sql

Purpose

Uses Hibernate to generate SQL to create Quartz database tables. The general format is:

```
$ grails create-jdbc-sql [FILENAME]
```

Examples

```
$ grails create-jdbc-sql quartz_jdbc.sql
```

Description

Creates in-memory Hibernate model objects and returns the SQL generated from those; this is similar to the process used by the schema-export script. The hibernate-core jar or the hibernate plugin must be in the classpath to use this command.