Quartz Scheduler Configuration Guide

Version 2.2.3

April 2016

This document applies to Quartz Scheduler Version 2.2.3 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-CG-223-20160415

Table of Contents

1 Configuration Overview

About Quartz Configuration

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

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

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

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

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

Configuration Topics

The properties for configuring various aspect of a scheduler are described in the following topics:

- Main Scheduler Configuration Settings - Configure primary scheduler settings, transactions.
- ThreadPool Configuration - Tune resources for job execution.
- Global Listener Configuration - Configuration of Listeners - Configure an application to receive notification of scheduled events.
- Plugin Configuration - Add functionality to your scheduler.
- RMI Configuration - Configuration of RMI Server and Client - Use a Quartz instance from a remote process.
- RAMJobStore Configuration - Store jobs and triggers in memory.
- JDBC JobStoreTX Configuration - Store jobs and triggers in a database via JDBC.

- JDBC JobStoreCMT Configuration - Configure JDBC with JTA-container-managed transactions.
- Configuring DataSources - Configure DataSource for use by the JDBC-JobStores.
- Cluster Configuration - Achieve fail-over and load-balancing with JDBC-JobStore.
- TerracottaJobStore Configuration - Clustering without a database.

2 Configuring the Main Scheduler Settings

Main Scheduler Configuration Settings

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

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

Property Name	Req'd	Type	Default Value
org.quartz.scheduler. wrapJobExecutionInUserTransaction	no	boolean	false
org.quartz.scheduler.skipUpdateCheck	no	boolean	false
org.quartz.scheduler. batchTriggerAcquisitionMaxCount	no	int	1
org.quartz.scheduler. batchTriggerAcquisitionFireAheadTimeWindow	no	long	0

org.quartz.scheduler.instanceName

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

org.quartz.scheduler.instanceId

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

org.quartz.scheduler.instanceIdGenerator.class

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

org.quartz.scheduler.threadName

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

org.quartz.scheduler.makeSchedulerThreadDaemon

A Boolean value (true or false) that specifies whether the main thread of the scheduler should be a daemon thread or not. See also the `org.quartz.threadPool.makeThreadsDaemons` property for tuning the

SimpleThreadPool in "ThreadPool Configuration" on page 16 if that is the thread pool implementation you are using (which is usually the case).

org.quartz.scheduler.threadsInheritContextClassLoaderOfInitializer

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

org.quartz.scheduler.idleWaitTime

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

org.quartz.scheduler.dbFailureRetryInterval

The amount of time in milliseconds that the scheduler will wait between re-tries after it detects a loss of connectivity within the JobStore (i.e., to the database). This parameter is not meaningful when using RamJobStore.

org.quartz.scheduler.classLoadHelper.class

Defaults to the most robust approach, which is to use the org.quartz.simpl.CascadingClassLoadHelper class, which in turn uses every other ClassLoadHelper class until one works. In general, you should not need to use any other class for this property. However, it might be useful for resolving class-loading issues on application servers. All of the current possible ClassLoadHelper implementations can be found in the org.quartz.simpl package.

org.quartz.scheduler.jobFactory.class

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

org.quartz.context.key.SOME_KEY

A name-value pair that will be placed into the scheduler context as strings (see Scheduler.getContext()). For example, setting the `org.quartz.context.key.MyKey`

to "MyValue" would perform the equivalent of scheduler.getContext().put("MyKey", "MyValue").

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

org.quartz.scheduler.userTransactionURL

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

org.quartz.scheduler.wrapJobExecutionInUserTransaction

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

org.quartz.scheduler.skipUpdateCheck

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

org.quartz.scheduler.batchTriggerAcquisitionMaxCount

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

org.quartz.scheduler.batchTriggerAcquisitionFireAheadTimeWindow

The amount of time (in milliseconds) that a trigger is allowed to be acquired and fired ahead of its scheduled fire time. Defaults to 0. The larger the number, the more likely batch acquisition of triggers to fire will be able to select and fire more than one trigger at a time. However, this comes at the cost of the trigger schedule not being honored precisely (triggers might fire early). This property can be useful (for the sake of

performance) in situations where the scheduler has a very large number of triggers that need to be fired at or near the same time.

3 Configuring the Threadpool

ThreadPool Configuration

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

org.quartz.threadPool.class

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

org.quartz.threadPool.threadCount

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

org.quartz.threadPool.threadPriority

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

SimpleThreadPool-Specific Properties

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

Property Name	Required	Type	Default Value
org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread	no	boolean	false
org.quartz.threadPool.threadNamePrefix	no	string	[Scheduler Name]_Worker

org.quartz.threadPool.makeThreadsDaemons

Set to true to have the threads in the pool created as daemon threads. Default is false. See also the `org.quartz.scheduler.makeSchedulerThreadDaemon` property in "Main Scheduler Configuration Settings" on page 10.

org.quartz.threadPool.threadsInheritGroupOfInitializingThread

Can be true or false. Defaults to true.

org.quartz.threadPool.threadsInheritContextClassLoaderOfInitializingThread

Can be true or false. Defaults to false.

org.quartz.threadPool.threadNamePrefix

The prefix for thread names in the worker pool. It will be postpended with a number.

Custom ThreadPools

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

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

4 Configuring Global Listeners

Global Listener Configuration

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

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

Configuring a Global TriggerListener

The general pattern for defining a global TriggerListener is:

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

Configuring a Global JobListener

The general pattern for defining a global JobListener is:

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

5 Configuring Plugins

Plugin Configuration

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

The general pattern for defining a plug-in is:

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

There are several Plugins that come with Quartz, that can be found in the org.quartz.plugins package (and subpackages). Examples of configuring a few of them are shown in the topics that follow.

Sample configuration of Logging Trigger History Plugin

The Logging Trigger History plugin catches trigger events (it is also a trigger listener) and logs them with Jakarta Commons-Logging. See the class's Javadoc for a list of all the possible parameters.

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

Sample configuration of XML Scheduling Data Processor Plugin

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

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

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

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

Sample configuration of Shutdown Hook Plugin

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

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

6 Configuring RMI Settings

RMI Configuration

None of the primary properties are required, and all have reasonable defaults. When using Quartz via RMI, you need to start an instance of Quartz with it configured to "export" its services via RMI. You then create clients to the server by configuring a Quartz scheduler to "proxy" its work to the server.

Note: Some users experience problems with class availability (namely Job classes) between the client and server. To work through these problems you'll need an understanding of the RMI codebase and security managers. You may find the resources to be useful:

The description of RMI and codebase at http://www.kedwards.com/jini/codebase.html. (One of the important points is to realize that codebase is used by the client.)

The Java API docs on the RMISecurityManager.

Property Name	Required	Default Value
org.quartz.scheduler.rmi.export	no	false
org.quartz.scheduler.rmi.registryHost	no	'localhost'
org.quartz.scheduler.rmi.registryPort	no	1099
org.quartz.scheduler.rmi.createRegistry	no	'never'
org.quartz.scheduler.rmi.serverPort	no	random
org.quartz.scheduler.rmi.proxy	no	false

org.quartz.scheduler.rmi.export

Set to true if you want the Quartz Scheduler to export itself via RMI as a server.

org.quartz.scheduler.rmi.registryHost

The host at which the RMI Registry can be found (often localhost).

org.quartz.scheduler.rmi.registryPort

The port on which the RMI Registry is listening (usually 1099).

org.quartz.scheduler.rmi.createRegistry

Specifies how you want Quartz to cause the creation of an RMI Registry.

Use "false" or "never" if you don't want Quartz to create a registry (e.g., if you already have an external registry running).

Use "true" or "as_needed" if you want Quartz to first attempt to use an existing registry, and then fall back to creating one.

Use "always" if you want Quartz to attempt creating a Registry, and then fall back to using an existing one.

If a registry is created, it will be bound to port number in the given `org.quartz.scheduler.rmi.registryPort` property and `org.quartz.rmi.registryHost` should be "localhost".

org.quartz.scheduler.rmi.serverPort

The port on which the Quartz Scheduler service will bind and listen for connections. By default, the RMI service will randomly select a port as the scheduler is bound to the RMI Registry.

org.quartz.scheduler.rmi.proxy

To connect to (use) a remotely served scheduler, set the `org.quartz.scheduler.rmi.proxy` flag to true. You must also then specify a host and port for the RMI Registry process - which is typically 'localhost' port 1099.

Note: It does not make sense to set both `org.quartz.scheduler.rmi.export` and `org.quartz.scheduler.rmi.proxy`to true in the same config file. If you do, the `export` property will be ignored. Setting both properties to false is, of course, valid if you are not using Quartz via RMI.

7 Configuring the RAMJobStore

RAMJobStore Configuration

RAMJobStore is used to store scheduling information (job, triggers and calendars) within memory. RAMJobStore is fast and lightweight, but all scheduling information is lost when the process terminates.

RAMJobStore is selected by setting the `org.quartz.jobStore.class` property as shown below:

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

RAMJobStore can be tuned with the following properties:

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

org.quartz.jobStore.misfireThreshold

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

8 Configuring JDBC JobStoreTX

JDBC JobStoreTX Configuration

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

JobStoreTX manages all transactions itself by calling commit() (or rollback()) on the database connection after every action (such as the addition of a job). JDBCJobStore is appropriate if you are using Quartz in a stand-alone application or within a servlet container if the application is not using JTA transactions.

The JobStoreTX is selected by setting the `org.quartz.jobStore.class` property as such:

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

JobStoreTX can be tuned with the following properties:

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

Property Name	Required	Type	Default Value
			LOCK_NAME = ? FOR UPDATE"
org.quartz.jobStore.txIsolationLevelSerializable	no	boolean	false
org.quartz.jobStore.acquireTriggersWithinLock	no	boolean	false (or true - see doc below)
org.quartz.jobStore.lockHandler.class	no	string	null
org.quartz.jobStore.driverDelegateInitString	no	string	null

org.quartz.jobStore.driverDelegateClass

Driver delegates understand the dialects of various database systems. Possible choices include:

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

Note that many databases are known to work with the StdJDBCDelegate, while others are known to work with delegates for other databases. Derby, for example, works well with the Cloudscape delegate.

org.quartz.jobStore.dataSource

The value of this property must be the name of one the DataSources defined in the configuration properties file. For more information, see "DataSource Configuration" on page 42.

org.quartz.jobStore.tablePrefix

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

org.quartz.jobStore.useProperties

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

org.quartz.jobStore.misfireThreshold

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

org.quartz.jobStore.isClustered

Set to true in order to turn on clustering features. This property must be set to rue if you are having multiple instances of Quartz use the same set of database tables. Otherwise data corruption and erratic behavior will result. For more information, see "Cluster Configuration" on page 48.

org.quartz.jobStore.clusterCheckinInterval

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

org.quartz.jobStore.maxMisfiresToHandleAtATime

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

org.quartz.jobStore.dontSetAutoCommitFalse

Setting this parameter to true tells Quartz not to call setAutoCommit(false) on connections obtained from the DataSource(s). This can be helpful in a few situations, such as if you have a driver that complains if it is called when it is already off. This

property defaults to false, because most drivers require that setAutoCommit(false) is called.

org.quartz.jobStore.selectWithLockSQL

A SQL string that selects a row in the LOCKS table and places a lock on the row. If not set, the default is "SELECT * FROM {0}LOCKS WHERE SCHED_NAME = {1} AND LOCK_NAME = ? FOR UPDATE", which works for most databases. The "{0}" is replaced at run time with the TABLE_PREFIX that you configured above. The "{1}" is replaced with the scheduler's name.

org.quartz.jobStore.txIsolationLevelSerializable

A value of true tells Quartz (when using JobStoreTX or CMT) to call setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE) on JDBC connections. This can be helpful to prevent lock timeouts with some databases under high loads and running long-lasting transactions.

org.quartz.jobStore.acquireTriggersWithinLock

Whether or not the acquisition of next triggers to fire should occur within an explicit database lock. This was once necessary (in previous versions of Quartz) to avoid deadlocks with particular databases, but is no longer considered necessary. Hence the default value is false.

If `org.quartz.scheduler.batchTriggerAcquisitionMaxCount` is > 1 and JDBC JobStore is used, then this property must be set to true to avoid data corruption (as of Quartz 2.1.1, true is the default when `batchTriggerAcquisitionMaxCount` is > 1).

org.quartz.jobStore.lockHandler.class

The class name to be used to produce an instance of a org.quartz.impl.jdbcjobstore.Semaphore to be used for locking control on the job store data. This is an advanced configuration feature, which should not be used by most users. By default, Quartz will select the most appropriate (pre-bundled) Semaphore implementation to use. "org.quartz.impl.jdbcjobstore.UpdateLockRowSemaphore" QUARTZ-497 may be of interest to MS SQL Server users. See QUARTZ-441.

org.quartz.jobStore.driverDelegateInitString

A pipe-delimited list of properties (and their values) that can be passed to the DriverDelegate during initialization time.

The format of the string is as follows:

```
"settingName=settingValue|otherSettingName=otherSettingValue|..."
```

The StdJDBCDelegate and all of its descendants (all delegates that ship with Quartz) support a property called `triggerPersistenceDelegateClasses`, which can be set to a comma-separated list of classes that implement the TriggerPersistenceDelegate interface for storing custom trigger types. See the Java classes SimplePropertiesTriggerPersistenceDelegateSupport and

SimplePropertiesTriggerPersistenceDelegateSupport for examples of writing a persistence delegate for a custom trigger.

9 Configuring JDBC JobStoreCMT

JDBC JobStoreCMT Configuration

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

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

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

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

JobStoreCMT can be tuned using the following properties:

Property Name	Required	Type	Default Value
org.quartz.jobStore.driverDelegateClass	yes	string	null
org.quartz.jobStore.dataSource	yes	string	null
org.quartz.jobStore. nonManagedTXDataSource	yes	string	null
org.quartz.jobStore.tablePrefix	no	string	"QRTZ_"
org.quartz.jobStore.useProperties	no	boolean	false
org.quartz.jobStore.misfireThreshold	no	int	60000
org.quartz.jobStore.isClustered	no	boolean	false
org.quartz.jobStore.clusterCheckinInterval	no	long	15000
org.quartz.jobStore. maxMisfiresToHandleAtATime	no	int	20

Property Name	Required	Type	Default Value
org.quartz.jobStore. dontSetAutoCommitFalse	no	boolean	false
org.quartz.jobStore. dontSetNonManagedTXConnection AutoCommitFalse	no	boolean	false
org.quartz.jobStore.selectWithLockSQL	no	string	"SELECT * FROM {0}LOCKS WHERE SCHED_NAME = {1} AND LOCK_NAME = ? FOR UPDATE"
org.quartz.jobStore. txIsolationLevelSerializable	no	boolean	false
org.quartz.jobStore. txIsolationLevelReadCommitted	no	boolean	false
org.quartz.jobStore. acquireTriggersWithinLock	no	boolean	false (or true - see doc below)
org.quartz.jobStore.lockHandler.class	no	string	null
org.quartz.jobStore.driverDelegateInitString	no	string	null

org.quartz.jobStore.driverDelegateClass

Driver delegates understand the dialects of various database systems. Possible choices include:

- org.quartz.impl.jdbcjobstore.StdJDBCDelegate (for fully JDBC-compliant drivers)
- org.quartz.impl.jdbcjobstore.MSSQLDelegate (for Microsoft SQL Server, and Sybase)
- org.quartz.impl.jdbcjobstore.PostgreSQLDelegate
- org.quartz.impl.jdbcjobstore.WebLogicDelegate (for WebLogic drivers)
- org.quartz.impl.jdbcjobstore.oracle.OracleDelegate

- org.quartz.impl.jdbcjobstore.oracle.WebLogicOracleDelegate (for Oracle drivers used within WebLogic)
- org.quartz.impl.jdbcjobstore.oracle.weblogic.WebLogicOracleDelegate (for Oracle drivers used within WebLogic)
- org.quartz.impl.jdbcjobstore.CloudscapeDelegate
- org.quartz.impl.jdbcjobstore.DB2v6Delegate
- org.quartz.impl.jdbcjobstore.DB2v7Delegate
- org.quartz.impl.jdbcjobstore.DB2v8Delegate
- org.quartz.impl.jdbcjobstore.HSQLDBDelegate
- org.quartz.impl.jdbcjobstore.PointbaseDelegate
- org.quartz.impl.jdbcjobstore.SybaseDelegate

Note that many databases are known to work with the StdJDBCDelegate, while others are known to work with delegates for other databases. Derby, for example, works well with the Cloudscape delegate.

org.quartz.jobStore.dataSource

The value of this property must be the name of one the DataSources defined in the configuration properties file. For JobStoreCMT, it is required that this DataSource contains connections that are capable of participating in JTA (i.e., container-managed) transactions. This typically means that the DataSource will be configured and maintained within and by the application server, and Quartz will obtain a handle to it via JNDI. For more information, see "DataSource Configuration" on page 42.

org.quartz.jobStore.nonManagedTXDataSource

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

org.quartz.jobStore.tablePrefix

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

org.quartz.jobStore.useProperties

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

org.quartz.jobStore.misfireThreshold

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

org.quartz.jobStore.isClustered

Set to true in order to turn on clustering features. This property must be set to true if you are having multiple instances of Quartz use the same set of database tables. Otherwise data corruption and erratic behavior will occur. For more information, see the configuration docs for clustering.

org.quartz.jobStore.clusterCheckinInterval

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

org.quartz.jobStore.maxMisfiresToHandleAtATime

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

org.quartz.jobStore.dontSetAutoCommitFalse

Setting this parameter to true tells Quartz not to call setAutoCommit(false) on connections obtained from the DataSource(s). This can be helpful in a few situations, such as when you have a driver that complains if it is called when it is already off. This property defaults to false, because most drivers require that setAutoCommit(false) is called.

org.quartz.jobStore.dontSetNonManagedTXConnectionAutoCommitFalse

The same as the property `org.quartz.jobStore.dontSetAutoCommitFalse` except that it applies to the nonManagedTXDataSource.

org.quartz.jobStore.selectWithLockSQL

Must be a SQL string that selects a row in the LOCKS table and places a lock on the row. If not set, the default is "SELECT * FROM {0}LOCKS WHERE SCHED_NAME = {1} AND LOCK_NAME = ? FOR UPDATE." This query string works for most databases. The "{0}" is replaced at run time with the TABLE_PREFIX that you configured above. The "{1}" is replaced with the scheduler's name.

org.quartz.jobStore.txIsolationLevelSerializable

A value of true tells Quartz to call setTransactionIsolation(Connection.TRANSACTION_SERIALIZABLE) on JDBC

connections. This can be helpful to prevent lock timeouts with some databases under high loads and running long-lasting transactions.

org.quartz.jobStore.txIsolationLevelReadCommitted

When set to true, this property tells Quartz to call setTransactionIsolation(Connection.TRANSACTION_READ_COMMITTED) on the non-managed JDBC connections. This can be helpful to prevent lock timeouts with some databases (such as DB2) under high loads and running long-lasting transactions.

org.quartz.jobStore.acquireTriggersWithinLock

Whether or not the acquisition of next triggers to fire should occur within an explicit database lock. This was once necessary (in previous versions of Quartz) to avoid dead-locks with particular databases, but is no longer considered necessary. Hence the default value is false.

If `org.quartz.scheduler.batchTriggerAcquisitionMaxCount` is > 1 and JDBC JobStore is used, then this property must be set to true to avoid data corruption (as of Quartz 2.1.1, true is the default when `batchTriggerAcquisitionMaxCount` is > 1).

org.quartz.jobStore.lockHandler.class

The class name to be used to produce an instance of a org.quartz.impl.jdbcjobstore.Semaphore to be used for locking control on the job store data. This is an advanced configuration feature, which should not be used by most users. By default, Quartz will select the most appropriate (pre-bundled) Semaphore implementation to use. "org.quartz.impl.jdbcjobstore.UpdateLockRowSemaphore" QUARTZ-497 may be of interest to MS SQL Server users. "JTANonClusteredSemaphore" which is bundled with Quartz may give improved performance when using JobStoreCMT, though it is an experimental implementation. See QUARTZ-441 and QUARTZ-442.

org.quartz.jobStore.driverDelegateInitString

A pipe-delimited list of properties (and their values) that can be passed to the DriverDelegate during initialization.

The format of the string is as follows:

```
"settingName=settingValue|otherSettingName=otherSettingValue|..."
```

The StdJDBCDelegate and all of its descendants (all delegates that ship with Quartz) support a property called `triggerPersistenceDelegateClasses`, which can be set to a comma-separated list of classes that implement the TriggerPersistenceDelegate interface for storing custom trigger types. See the Java classes SimplePropertiesTriggerPersistenceDelegateSupport and SimplePropertiesTriggerPersistenceDelegateSupport for examples of writing a persistence delegate for a custom trigger.

10 Configuring DataSources

DataSource Configuration

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

DataSources can be configured in three ways:

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

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

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

Quartz DataSource Properties

Quartz-created DataSources are defined using the following properties:

Property Name	Required	Type	Default Value
org.quartz.dataSource.NAME.driver	yes	String	null
org.quartz.dataSource.NAME.URL	yes	String	null
org.quartz.dataSource.NAME.user	no	String	""
org.quartz.dataSource.NAME.password	no	String	""
org.quartz.dataSource.NAME. maxConnections	no	int	10

Property Name	Required	Type	Default Value
org.quartz.dataSource.NAME.validationQuery	no	String	null
org.quartz.dataSource.NAME.idleConnectionValidationSeconds	no	int	50
org.quartz.dataSource.NAME.validateOnCheckout	no	boolean	false
org.quartz.dataSource.NAME.discardIdleConnectionsSeconds	no	int	0 (disabled)

org.quartz.dataSource.NAME.driver

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

org.quartz.dataSource.NAME.URL

The connection URL (host, port, and so forth) for connection to your database.

org.quartz.dataSource.NAME.user

The user name to use when connecting to your database.

org.quartz.dataSource.NAME.password

The password to use when connecting to your database.

org.quartz.dataSource.NAME.maxConnections

The maximum number of connections that the DataSource can create in its pool of connections.

org.quartz.dataSource.NAME.validationQuery

An optional SQL query string that the DataSource can use to detect and replace failed/corrupt connections. For example a user of an Oracle database might use "select table_name from user_tables," which is a query that should not fail unless the connection is actually bad.

org.quartz.dataSource.NAME.idleConnectionValidationSeconds

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

org.quartz.dataSource.NAME.validateOnCheckout

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

org.quartz.dataSource.NAME.discardIdleConnectionsSeconds

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

Example of a Quartz-defined DataSource

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

Application Server DataSource Properties

DataSources that are managed by your application server are defined using the following properties:

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

org.quartz.dataSource.NAME.jndiURL

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

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

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

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

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

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

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

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

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

Example of a DataSource referenced from an Application Server

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

Custom ConnectionProvider Implementations

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

org.quartz.dataSource.NAME.connectionProvider.class

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

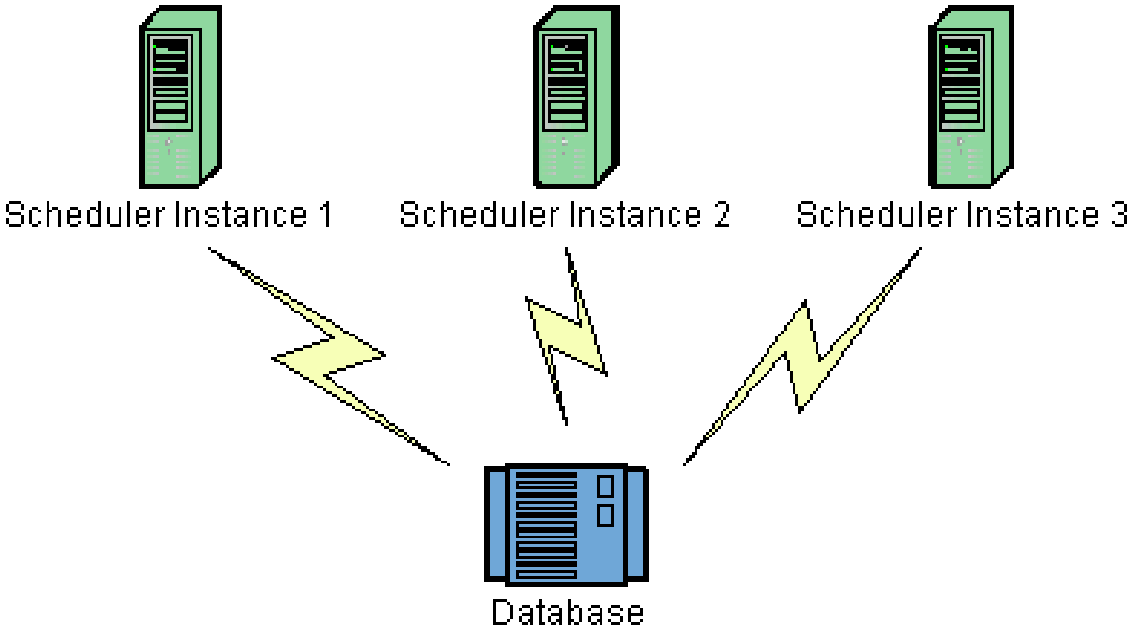
Following is an example of using a custom ConnectionProvider implementation:

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

11 Configuring Clustering

Cluster Configuration

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



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

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

Only one node will fire the job for each firing. Meaning that, if the job has a repeating trigger that is set 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, and so forth. It won't necessarily be the same node each time. The selected node is chosen more or less at random. The load balancing mechanism is near-random for busy schedulers (lots of triggers) but favors the same node for non-busy (few triggers) schedulers.

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

The clustering feature works best for scaling out long-running and/or CPU-intensive jobs (i.e., distributing the work-load over multiple nodes). If you need to scale out to support thousands of short-running (e.g., one second) jobs, consider partitioning

the set of jobs by using multiple distinct schedulers (including multiple clustered schedulers for HA). The scheduler makes use of a cluster-wide lock, a pattern that degrades performance as you add more nodes (when going beyond about three nodes - depending upon your database's capabilities, etc.).

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

Important: Never run clustering on separate machines, unless their clocks are synchronized using some form of time-sync service (daemon) that runs very regularly (the clocks must be within a second of each other). See http://www.boulder.nist.gov/timefreq/service/its.htm if you are unfamiliar with how to do this.

Important: Never start (scheduler.start()) a non-clustered instance against the same set of database tables that any other instance is running (start()ed) against. Doing so will result in data corruption and erratic behavior.

Example Properties For A Clustered Scheduler

```
#============================================================================
# Configure Main Scheduler Properties
#============================================================================
org.quartz.scheduler.instanceName = MyClusteredScheduler
org.quartz.scheduler.instanceId = AUTO
#============================================================================
# Configure ThreadPool
#============================================================================
org.quartz.threadPool.class = org.quartz.simpl.SimpleThreadPool
org.quartz.threadPool.threadCount = 25
org.quartz.threadPool.threadPriority = 5
#============================================================================
# Configure JobStore
#============================================================================
org.quartz.jobStore.misfireThreshold = 60000
org.quartz.jobStore.class = org.quartz.impl.jdbcjobstore.JobStoreTX
org.quartz.jobStore.driverDelegateClass =
org.quartz.impl.jdbcjobstore.oracle.OracleDelegate
org.quartz.jobStore.useProperties = false
org.quartz.jobStore.dataSource = myDS
org.quartz.jobStore.tablePrefix = QRTZ _
org.quartz.jobStore.isClustered = true
org.quartz.jobStore.clusterCheckinInterval = 20000
#============================================================================
# Configure Datasources
#============================================================================
org.quartz.dataSource.myDS.driver = oracle.jdbc.driver.OracleDriver
org.quartz.dataSource.myDS.URL = jdbc:oracle:thin:@polarbear:1521:dev
org.quartz.dataSource.myDS.user = quartz
org.quartz.dataSource.myDS.password = quartz
org.quartz.dataSource.myDS.maxConnections = 5
```

```
org.quartz.dataSource.myDS.validationQuery=select 0 from dual
```

12 Configuring the TerracottaJobStore

TerracottaJobStore Configuration

TerracottaJobStore is used to store scheduling information (job, triggers and calendars) within a Terracotta server.

TerracottaJobStore is much more performant than utilizing a database for storing scheduling data (via JDBC-JobStore), and yet offers clustering features such as load-balancing and fail-over.

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

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

TerracottaJobStore is selected by setting the `org.quartz.jobStore.class` property as shown below:

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

TerracottaJobStore can be tuned with the following properties:

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

org.quartz.jobStore.tcConfigUrl

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

org.quartz.jobStore.misfireThreshold

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