Terracotta Quartz User Guide

Version 2.2.1

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

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

Document ID: QS-TQUG-221-20141015

Table of Contents

1 Using Terracotta Quartz Scheduler

What is Terracotta Quartz Scheduler?

Terracotta Quartz Scheduler is a licensed version of Quartz Scheduler that includes the TerracottaJobStore and enables you to cluster the Quartz Scheduler service using the Terracotta Server Array. Using Terracotta Quartz Scheduler provides a number of advantages:

- **Adds High-Availability** – "hot" standbys can immediately replace failed servers with no downtime, no lost data.
- **Provides a Locality API** – Route jobs to specific node groups or base routing decisions on system characteristics and available resources.
- **Improves Performance** – Offload traditional databases and automatically distribute load.
- **Provides a Clear Scale-Out Path** – Add capacity without requiring additional database resources.
- **Ensures Persistence** – Automatically back up shared data without impacting cluster performance.

Adding Terracotta Clustering to Quartz Scheduler

This document describes how to add Terracotta clustering to an application that is using Quartz Scheduler. Use this installation if you have been running your Quartz Scheduler application:

- on a single JVM, or
- on a cluster using JDBC-Jobstore.

To set up the cluster with Terracotta, you will add a Terracotta JAR to each application and run a Terracotta Server Array. Except as noted in this document, you can continue to use Quartz in your application as specified in the Quartz documentation.

Prerequisites

- JDK 1.6 or higher.
- BigMemory Max 4.0.2 or higher. Download the kit and run the installer on the machine that will host the Terracotta Server.
- All clustered Quartz objects must be serializable. For example, if you create Quartz objects such as Trigger types, they must be serializable.

Step 1: Install Quartz Scheduler

For guaranteed compatibility, use the JAR files included with the Terracotta kit you are installing. Mixing with older components may cause errors or unexpected behavior.

To install the Quartz Scheduler in your application, add the following JAR files to your application's classpath:

- `${TERRACOTTA_HOME}/quartz/quartz-ee-<quartz-version>.jar`

  <quartz-version> is the current version of Quartz (2.2.0 or higher).

- `${TERRACOTTA_HOME}/common/terracotta-toolkit-runtime-ee-<version>.jar`

  The Terracotta Toolkit JAR contains the Terracotta client libraries. <version> is the current version of the Terracotta Toolkit JAR (4.0.2 or higher).

If you are using a WAR file, add these JAR files to its `WEB-INF/lib` directory.

Most application servers (or web containers) should work with this installation of the Quartz Scheduler. However, note the following:

- GlassFish application server – You must add `<jvm-options>-Dcom.sun.enterprise.server.ss.ASQuickStartup=false</jvm-options>` to domains.xml.
- WebLogic application server – You must use the supported versions of WebLogic. See support information on https://confluence.terracotta.org/display/release/Home.

Step 2: Configure Quartz Scheduler

The Quartz configuration file, quartz.properties, by default, should be on your application's classpath. If you are using a WAR file, add the Quartz configuration file to WEB-INF/classes or to a JAR file that is included in WEB-INF/lib.

Add Terracotta Configuration

To be clustered by Terracotta, the following properties in quartz.properties must be set as follows:

```
# If you use the jobStore class TerracottaJobStore,
# Quartz Where will not be available.
org.quartz.jobStore.class = org.terracotta.quartz.EnterpriseTerracottaJobStore
org.quartz.jobStore.tcConfigUrl = <path/to/Terracotta/configuration>
```

The property `org.quartz.jobStore.tcConfigUrl` must point the client (or application server) at the location of the Terracotta configuration.

Note: In a Terracotta cluster, the application server is also known as the *client*.

The client must load the configuration from a file or a Terracotta server. If loading from a server, give the server's hostname and its tsa-port (9510 by default), found in the Terracotta configuration. The following example shows a configuration that is loaded from the Terracotta server on the local host:

```
# If you use the jobStore class TerracottaJobStore,
# Quartz Where will not be available.
org.quartz.jobStore.class = org.terracotta.quartz.EnterpriseTerracottaJobStore
org.quartz.jobStore.tcConfigUrl = localhost:9510
```

To load Terracotta configuration from a Terracotta configuration file (tc-config.xml by default), use a path. For example, if the Terracotta configuration file is located on

myHost.myNet.net at /usr/local/TerracottaHome, use the full URI along with the configuration file's name:

```
# If you use the jobStore class TerracottaJobStore,
# Quartz Where will not be available.
org.quartz.jobStore.class = org.terracotta.quartz.EnterpriseTerracottaJobStore
org.quartz.jobStore.tcConfigUrl =
            file://myHost.myNet.net/usr/local/TerracottaHome/tc-config.xml
```

If the Terracotta configuration source changes at a later time, it must be updated in configuration.

Scheduler Instance Name

A Quartz scheduler has a default name configured by the following quartz.properties property:

```
org.quartz.scheduler.instanceName = QuartzScheduler
```

Setting this property is not required. However, you can use this property to instantiate and differentiate between two or more instances of the scheduler, each of which then receives a separate store in the Terracotta cluster.

Using different scheduler names allows you to isolate different job stores within the Terracotta cluster (logically unique scheduler instances). Using the same scheduler name allows different scheduler instances to share the same job store in the cluster.

Step 3: Start the Cluster

1. Start the Terracotta server:

   On UNIX/Linux

   ```
   [PROMPT] ${TERRACOTTA_HOME}/server/bin/start-tc-server.sh &
   ```

   On Microsoft Windows

   ```
   [PROMPT] ${TERRACOTTA_HOME}\server\bin\start-tc-server.bat
   ```

2. Start the application servers.
3. To monitor the servers, start the Terracotta Management Console:

   On UNIX/Linux

   ```
   [PROMPT] ${TERRACOTTA_HOME}/tools/management-console/bin/start-tmc.sh &
   ```

   On Microsoft Windows

   ```
   [PROMPT] ${TERRACOTTA_HOME}\tools\management-console\bin\start-tmc.bat
   ```

Step 4: Edit the Terracotta Configuration

This step shows you how to run clients and servers on separate machines and add failover (High Availability). You will expand the Terracotta cluster and add High Availability by doing the following:

- Moving the Terracotta server to its own machine
- Creating a cluster with multiple Terracotta servers

- Creating multiple application nodes

These tasks bring your cluster closer to a production architecture.

Procedure:

1. Shut down the Terracotta cluster.

   On UNIX/Linux

   ```
   [PROMPT] ${TERRACOTTA_HOME}/server/bin/stop-tc-server.sh
   ```

   On Microsoft Windows

   ```
   [PROMPT] ${TERRACOTTA_HOME}\server\bin\stop-tc-server.bat
   ```

2. Create a Terracotta configuration file called tc-config.xml with contents similar to the following:

```
<?xml version="1.0" encoding="UTF-8"?>
<con:tc-config xmlns:con="http://www.terracotta.org/config">
  <servers>
    <mirror-group group-name="default-group">
       <!-- Sets where the Terracotta server can be found.
        Replace the value of host with the server's IP address. -->
      <server host="%i" name="server1">
        <offheap>
          <enabled>true</enabled>
          <maxDataSize>512M</maxDataSize>
        </offheap>
        <tsa-port>9510</tsa-port>
        <jmx-port>9520</jmx-port>
        <data>terracotta/data</data>
        <logs>terracotta/logs</logs>
        <data-backup>terracotta/backups</data-backup>
      </server>
       <!-- If using a mirror Terracotta server, also referred to as an
        ACTIVE-PASSIVE configuration, add the second server here. -->
      <server host="%i" name="Server2">
        <offheap>
          <enabled>true</enabled>
          <maxDataSize>512M</maxDataSize>
        </offheap>
        <tsa-port>9511</tsa-port>
        <data>terracotta/data-dos</data>
        <logs>terracotta/logs-dos</logs>
        <data-backup>terracotta/backups-dos</data-backup>
      </server>
    </mirror-group>
    <update-check>
      <enabled>false</enabled>
    </update-check>
    <garbage-collection>
      <enabled>true</enabled>
    </garbage-collection>
    <restartable enabled="true"/>
  </servers>
  <!-- Sets where the generated client logs are saved on clients. -->
  <clients>
    <logs>terracotta/logs</logs>
  </clients>
</con:tc-config>
```

3. Install BigMemory Max on a separate machine for each server you configure in tc-config.xml.
4. Copy the tc-config.xml to a location accessible to the Terracotta servers.
5. Perform "Step 1: Install Quartz Scheduler" on page 6and "Step 2: Configure Quartz Scheduler" on page 7 on each application node you want to run in the cluster. Be sure to install your application and any application servers on each node.
6. Edit the `org.quartz.jobStore.tcConfigUrl` property in quartz.properties to list both Terracotta servers: org.quartz.jobStore.tcConfigUrl = <server.1.ip.address>:9510,<server.2.ip.address>:9510
7. Copy quartz.properties to each application node and ensure that it is on your application's classpath. If you are using a WAR file, add the Quartz configuration file to WEB-INF/classes or to a JAR file that is included in WEB-INF/lib.
8. Start the Terracotta server in the following way, replacing "Server1" with the name you gave your server in tc-config.xml:

   On UNIX/Linux

   ```
   [PROMPT] ${TERRACOTTA_HOME}/server/bin/start-tc-server.sh \
      -f <path/to/tc-config.xml> -n Server1 &
   ```

   On Microsoft Windows

   ```
   [PROMPT] ${TERRACOTTA_HOME}\server\bin\start-tc-server.bat ^
      -f <path\to\tc-config.xml> -n Server1 &
   ```

   If you configured a second server, start that server in the same way on its machine, entering its name after the `-n` flag. The second server to start up becomes the mirror. Any other servers you configured will also start up as mirrors.
9. Start all application servers.
10. Start the Terracotta Management Server and view the cluster.

Usage Notes

This topic contains information on functional aspects of Terracotta Quartz Scheduler and optimizing your use of TerracottaJobstore for Quartz Scheduler.

Execution of Jobs

In the general case, exactly one Quartz Scheduler node, or Terracotta client, executes a clustered job when that job's trigger fires. This can be any of the nodes that have the job. If a job repeats, it may be executed by any of the nodes that have it exactly once per the interval configured. It is not possible to predict which node will execute the job.

With Quartz Scheduler Where, a job can be assigned to a specific node based on certain criteria.

Working With JobDataMaps

JobDataMaps contain data that may be useful to jobs at execution time. A JobDataMap is stored at the time its associated job is added to a scheduler.

Updating a JobDataMap

If the stored job is stateful (meaning that it implements the StatefulJob interface), and the contents of its JobDataMap is updated (from within the job) during execution, then a new copy of the JobDataMap is stored when the job completes.

If the job is not stateful, then it must be explicitly stored again with the changed JobDataMap to update the stored copy of the job's JobDataMap. This is because TerracottaJobStore contains deep copies of JobDataMap objects and does not reflect updates made after a JobDataMap is stored.

Best Practices for Storing Objects in a JobDataMap

Because TerracottaJobStore contains deep copies of JobDataMap objects, application code should not have references to mutable JobDataMap objects. If an application relies on these references, there is risk of getting stale data as the mutable objects in a deep copy do not reflect changes made to the JobDataMap after it is stored.

To maximize performance and ensure long-term compatibility, place only Strings and primitives in JobDataMap. JobDataMap objects are serialized and prone to class-versioning issues. Putting complex objects into a clustered JobDataMap could also introduce other errors that are avoided with Strings and primitives.

Cluster Data Safety

By default, Terracotta clients (application servers) do not block to wait for a "transaction received" acknowledgment from a Terracotta server when writing data transactions to the cluster. This asynchronous write mode translates into better performance in a Terracotta cluster.

However, the option to maximize data safety by requiring an acknowledgment is available using the following Quartz configuration property:

```
org.quartz.jobStore.synchronousWrite = true
```

When `synchronousWrite` is set to "true", a client blocks with each transaction written to the cluster until an acknowledgment is received from a Terracotta server. This ensures that the transaction is committed in the cluster before the client continues work.

Effective Scaling Strategies

Clustering Quartz schedulers is an effective approach to distributing load over a number of nodes if jobs are long-running or are CPU intensive (or both). Distributing the jobs lessens the burden on system resources. In this case, and with a small set of jobs, lock contention is usually infrequent.

However, using a single scheduler forces the use of a cluster-wide lock, a pattern that degrades performance as you add more clients. The cost of this cluster-wide

lock becomes more evident if a large number of short-lived jobs are being fired by a single scheduler. In this case, consider partitioning the set of jobs across more than one scheduler.

If you do employ multiple schedulers, they can be run on every node, striping the cluster-wide locks. This is an effective way to reduce lock contention while adding scale.

If you intend to scale, measure your cluster's throughput in a test environment to discover the optimal number of schedulers and nodes.

2 Using Quartz Scheduler Where

What is Quartz Scheduler Where?

Quartz Scheduler Where is a capability provided by Terracotta Quartz Scheduler that allows you to control where jobs execute. Quartz Scheduler Where enables jobs and triggers to run on specified Terracotta clients instead of randomly chosen ones.

Quartz Scheduler Where provides a locality API that has a more readable fluent interface for creating and scheduling jobs and triggers. This locality API, together with configuration, can be used to route jobs to nodes based on defined criteria:

- Specific resources constraints such as free memory.
- Specific system characteristics such as type of operating system.
- A member of a specified group of nodes.

This document shows you how to configure and use the locality API. To use Quartz Scheduler Where, you should be familiar with Quartz Scheduler. For information about Quartz Scheduler, see the Quartz Scheduler documentation at http://quartz-scheduler.org/documentation.

Configuring Quartz Scheduler Where

To configure Quartz Scheduler Where, follow these steps:

1. Edit quartz.properties to cluster with Terracotta. For details, see "Adding Terracotta Clustering to Quartz Scheduler" on page 6.
2. If you intend to use node groups, configure an implementation of org.quartz.spi.InstanceIdGenerator to generate instance IDs to be used in the locality configuration. For information about generating instance IDs, see "Understanding Generated Node IDs" on page 15 .
3. Configure the node and trigger groups in quartzLocality.properties. For example:

```
# Set up node groups that can be referenced from application code.
# The values shown are instance IDs:
org.quartz.locality.nodeGroup.slowJobs = node0, node3
org.quartz.locality.nodeGroup.fastJobs = node1, node2
org.quartz.locality.nodeGroup.allNodes = node0, node1, node2, node3
# Set up trigger groups whose triggers fire only on nodes
# in the specified node groups. For example, a trigger in the
# trigger group slowTriggers will fire only on node0 and node3:
org.quartz.locality.nodeGroup.slowJobs.triggerGroups = slowTriggers
org.quartz.locality.nodeGroup.fastJobs.triggerGroups = fastTriggers
```

4. Ensure that quartzLocality.properties is on the classpath, the same as quartz.properties.

   See "Quartz Scheduler Where Code Sample" on page 19 for an example of how to use Quartz Scheduler Where.

Understanding Generated Node IDs

Terracotta clients each run an instance of a clustered Quartz Scheduler scheduler. Every instance of this clustered scheduler must use the same scheduler name, specified in quartz.properties. For example:

```
# Name the clustered scheduler.
org.quartz.scheduler.instanceName = myScheduler
```

myScheduler's data is shared across the cluster by each of its instances. However, every instance of myScheduler must also be identified uniquely, and this unique ID is specified in quartz.properties by the property `org.quartz.scheduler.instanceId`. This property should have one of the following values:

- A string value that identifies the scheduler instance running on the Terracotta client that loaded the containing quartz.properties. Each scheduler instance must have a unique ID value.
- AUTO – Delegates the generation of unique instance IDs to the class specified by the property `org.quartz.scheduler.instanceIdGenerator.class`.

For example, you can set `org.quartz.scheduler.instanceId` to "node1" on one node, "node2" on another node, and so on.

If you set `org.quartz.scheduler.instanceId` equal to "AUTO", then you should specify a generator class in quartz.properties using the property `org.quartz.scheduler.instanceIdGenerator.class`. This property can have one of the values listed in the following table.

Value	Notes
`org.quartz.simpl.HostnameInstanceIdGenerator`	Returns the hostname as the instance ID.
`org.quartz.simpl.SystemPropertyInstanceIdGenerator`	Returns the value of the `org.quartz.scheduler.instanceId` system property. Available with Quartz 2.0 or higher.
`org.quartz.simpl.SimpleInstanceIdGenerator`	Returns an instance ID composed of the local hostname with the current timestamp appended. Ensures a unique name. If you do not specify a generator class, this generator class is used by default. However, this class is not suitable for use with Quartz Scheduler Where

Value	Notes
	because the IDs it generates are not predictable.
Custom	Specify your own implementation of the interface org.quartz.spi.InstanceIdGenerator.

Using SystemPropertyInstanceIdGenerator

`org.quartz.simpl.SystemPropertyInstanceIdGenerator` is useful in environments that use initialization scripts or configuration files. For example, you could add the instanceId property to an application server's startup script in the form – `Dorg.quartz.scheduler.instanceId=node1`, where "node1" is the instance ID assigned to the local Quartz Scheduler scheduler. Or it could also be added to a configuration resource such as an XML file that is used to set up your environment.

The instanceId property values configured for each scheduler instance can be used in quartzLocality.properties node groups. For example, if you configured instance IDs node1, node2, and node3, you can use these IDs in node groups:

```
org.quartz.locality.nodeGroup.group1 = node1, node2
org.quartz.locality.nodeGroup.allNodes = node1, node2, node3
```

Available Constraints

Quartz Scheduler Where offers the following constraints:

- CPU – Provides methods for constraints based on minimum number of cores, available threads, and maximum amount of CPU load.
- Resident keys – Use a node with a specified BigMemory Max distributed cache that has the best match for the specified keys.
- Memory – Minimum amount of memory available.
- Node group – A node in the specified node group, as defined in quartzLocality.properties.
- OS – A node running the specified operating system.

To understand how to use these constraints, see the code samples provided in "Sample Code for Quartz Scheduler Where" on page 20.

Failure Scenarios

If a trigger cannot fire on the specified node or targeted node group, the associated job will not execute. Once the misfireThreshold timeout value is reached, the trigger misfires and any misfire instructions are executed.

Locality with the Standard Quartz Scheduler

It is also possible to add locality to jobs and triggers created with the standard Quartz Scheduler API by assigning the triggers to a trigger group specified in quartzLocality.properties.

3 Quartz Scheduler Where Code Sample

Sample Code for Quartz Scheduler Where

In this example, a cluster has Terracotta clients running Quartz Scheduler on the following hosts: node0, node1, node2, node3. These hostnames are used as the instance IDs for the Quartz Scheduler scheduler instances because the following properties are set in quartz.properties as shown here:

```
org.quartz.scheduler.instanceId = AUTO
#This sets the hostnames as instance IDs:
org.quartz.scheduler.instanceIdGenerator.class =
    org.quartz.simpl.HostnameInstanceIdGenerator
```

Setting quartzLocality Properties

quartzLocality.properties has the following configuration:

```
org.quartz.locality.nodeGroup.slowJobs = node0, node3
org.quartz.locality.nodeGroup.fastJobs = node1, node2
org.quartz.locality.nodeGroup.allNodes = node0, node1, node2, node3
org.quartz.locality.nodeGroup.slowJobs.triggerGroups = slowTriggers
org.quartz.locality.nodeGroup.fastJobs.triggerGroups = fastTriggers
```

Creating Locality-Aware Jobs and Triggers

The following code snippet uses Quartz Scheduler Where to create locality-aware jobs and triggers.

```
// Note the static imports of builder classes that define a Domain Specific Language (DSL).
import static org.quartz.JobBuilder.newJob;
import static org.quartz.TriggerBuilder.newTrigger;
import static org.quartz.locality.LocalityTriggerBuilder.localTrigger;
import static org.quartz.locality.NodeSpecBuilder.node;
import static org.quartz.locality.constraint.NodeGroupConstraint.partOfNodeGroup;
import org.quartz.JobDetail;
import org.quartz.locality.LocalityTrigger;
// Other required imports...
// Using the Quartz Scheduler fluent interface, or the DSL.
/***** Node Group + OS Constraint
Create a locality-aware job that can be run on any node
from nodeGroup "group1" that runs a Linux OS:
*****/
LocalityJobDetail jobDetail1 =
        localJob(
            newJob(myJob1.class)
                .withIdentity("myJob1")
                .storeDurably(true)
                .build())
            .where(
                node()
                    .is(partOfNodeGroup("group1"))
                    .is(OsConstraint.LINUX))
            .build();
// Create a trigger for myJob1:
Trigger trigger1 = newTrigger()
            .forJob("myJob1")
            .withIdentity("myTrigger1")
            .withSchedule(simpleSchedule()
                   .withIntervalInSeconds(10)
```

```
                    .withRepeatCount(2))
            .build();
// Create a second job:
JobDetail jobDetail2 = newJob(myJob2.class)
                .withIdentity("myJob2")
                .storeDurably(true)
                .build();
/***** Memory Constraint
Create a locality-aware trigger for myJob2 that will fire on any
node that has a certain amount of free memory available:
*****/
LocalityTrigger trigger2 =
        localTrigger(newTrigger()
            .forJob("myJob2")
            .withIdentity("myTrigger2"))
            .where(
                node()
                    // fire on any node in allNodes
        //  with at least 100MB in free memory.
                    .is(partOfNodeGroup("allNodes"))
                    .has(atLeastAvailable(100, MemoryConstraint.Unit.MB)))
            .build();
/***** A Locality-Aware Trigger For an Existing Job
 The following trigger will fire myJob1 on any node in the allNodes group
 that's running Linux:
*****/
LocalityTrigger trigger3 =
        localTrigger(newTrigger()
            .forJob("myJob1")
            .withIdentity("myTrigger3"))
            .where(
                node()
                    .is(partOfNodeGroup("allNodes")))
            .build();
/***** Locality Constraint Based on Cache Keys
The following job detail sets up a job (cacheJob) that will be fired on the node
where myCache has, locally, the most keys specified in the collection myKeys.
After the best match is found, missing elements will be faulted in.
If these types of jobs are fired frequently and a large amount of data must
often be faulted in, performance could degrade. To maintain performance, ensure
that most of the targeted data is already cached.
*****/
// myCache is already configured, populated, and distributed.
Cache myCache = cacheManager.getEhcache("myCache");
// A Collection is needed to hold the keys for elements targeted by cacheJob.
// The following assumes String keys.
Set<String> myKeys = new HashSet<String>();
... // Populate myKeys with the keys for the target elements in myCache.
// Create the job that will do work on the target elements.
LocalityJobDetail cacheJobDetail =
        localJob(
            newJob(cacheJob.class)
                .withIdentity("cacheJob")
                .storeDurably(true)
                .build())
                .where(
                    node()
                         .has(elements(myCache, myKeys)))
                .build();
```

Notice that trigger3, the third trigger defined, overrode the `partOfNodeGroup` constraint of myJob1. Where triggers and jobs have conflicting constraints, the triggers take priority. However, since trigger3 did not provide an OS constraint, it did *not* override

the OS constraint in myJob1. If any of the constraints in effect — trigger or job — are not met, the trigger will go into an error state and the job will not be fired.

Using CPU-Based Constraints

The CPU constraint allows you to run jobs on machines with adequate processing power:

```
...
import static org.quartz.locality.constraint.CpuConstraint.loadAtMost;
...
// Create a locality-aware trigger for someJob.
LocalityTrigger trigger =
        localTrigger(newTrigger()
            .forJob("someJob")
            .withIdentity("someTrigger"))
            .where(
                node()
                    // fire on any node in allNodes
                    // with at most the specified load:
                    .is(partOfNodeGroup("allNodes"))
                    .has(loadAtMost(.80)))
            .build();
```

The load constraint refers to the CPU load (a standard *NIX load measurement) averaged over the last minute. A load average below 1.00 indicates that the CPU is likely to execute the job immediately. The smaller the load, the freer the CPU, though setting a threshold that is too low could make it difficult for a match to be found.

Other CPU constraints include `CpuContraint.coresAtLeast(int amount)`, which specifies a node with a minimum number of CPU cores, and `CpuConstraint.threadsAvailableAtLeast(int amount)`, which specifies a node with a minimum number of available threads.

Note: If a trigger cannot fire because it has constraints that cannot be met by any node, that trigger will go into an error state. Applications using Quartz Scheduler Where with constraints should be tested under conditions that simulate those constraints in the cluster.

This example showed how memory and node-group constraints are used to route locality-aware triggers and jobs. For example, trigger2 is set to fire myJob2 on a node in a specific group (“allNodes”) with a specified minimum amount of free memory. A constraint based on operating system (Linux, Microsoft Windows, Apple OSX, and Oracle Solaris) is also available.