Configuration Guide and Reference

About Terracotta Documentation

This documentation is about Terracotta DSO, an advanced distributed-computing technology aimed at meeting special clustering requirements.

Terracotta products without the overhead and complexity of DSO meet the needs of almost all use cases and clustering requirements. To learn how to migrate from Terracotta DSO to standard Terracotta products, see Migrating From Terracotta DSO. To find documentation on non-DSO (standard) Terracotta products, see Terracotta Documentation. Terracotta release information, such as release notes and platform compatibility, is found in Product Information.

Release: 3.6
Publish Date: November, 2011 **Documentation Archive »**

Configuration Guide and Reference

Introduction

This document is a usage guide to the Terracotta configuration system and a reference to all of the Terracotta configuration elements.

Configuration Schema

The documentation for the Terracotta configuration XML document can be found here: http://www.terracotta.org/schema/

Reference Configuration

The reference configuration is a sample configuration file with inline comments describing each configuration element. The Terracotta kit contains a reference configuration file for the standard installation (`/config-samples/tc-config-express-reference.xml`) and DSO reference configuration file (`/platform/docs/tc-config-reference.xml`).

While these files serve as a good reference for the Terracotta configuration, do not start with either of these files as the basis for your Terracotta config. For that purpose, you should start with one of the sample configurations as described in the next section.

Sample Configurations

Sample configurations are provided in the Terracotta download kit. Typically the sample configurations are named `tc-config.xml` and can be found under the various samples provided with the Terracotta kit. We recommend starting with one of those configurations and modifying them to suit your needs.

Configuration Structure

The Terracotta configuration is an XML document that has four main sections: system, servers, clients, and application. Each of these sections provides a number of configuration options relevant to its particular configuration topic.

The Terracotta configuration is a single XML file. As of Terracotta 2.3, the configuration file can include additional configuration by referencing any number of Terracotta Integration Modules.

Terracotta Integration Modules

Terracotta Integration Modules (TIMs) allow sets of configuration elements to be packaged together as a single, includable module within your configuration. While a Terracotta configuration file resides with a Terracotta server instance, TIMs are never installed on Terracotta server instances. This is because applications are never integrated with Terracotta server instances, only with Terracotta clients. Terracotta clients get their TIM configurations when they fetch the Terracotta configuration file from a Terracotta server instance or by having their own Terracotta configuration files.

A number of configuration modules are provided in the Terracotta distribution for JDK library support, Tomcat session clustering support, and many others. New and updated configuration modules for third-party products are added regularly. You can choose from existing configuration modules, or create your own.

The following resources are available for more information on existing Terracotta Integration Modules, how they work, and how to build your own:

- Downloading Integration Modules
- Creating Terracotta Integration Modules
- Terracotta Forge

How to Use Integration Modules

The `modules` directory under the Terracotta installation directory contains a number of JAR files, each of which is an integration module.

To add an integration module, first identify the module JAR file, which has the following format:

```
<name-of-application>-<version.of.application>-<version.of.TIM>-SNAPSHOT.jar
```

The -SNAPSHOT portion of the version appears in a TIM filename only if the TIM is not a released version.

Add the the module's filename and version number to the `tc-config.xml` file by adding a `<clients>`/`<modules>`/`<module>` element as follows:

```
...
<clients>
...
  <modules>
  ...
    <module name="<name-of-application>-<version.of.application>" version="<version.of.TIM>" />
  ...
  </modules>
...
</clients>
...
```

Module Versions Are Optional

Since the tim-get script finds the optimal version for the current installation of the Terracotta kit, module versions are optional.

When you specify a `name` and `version` for a module, the Terracotta runtime searches for the matching `jar` file. If filenames are changed or files are moved after being specified in `tc-config.xml`, the modules cannot be loaded.

Using the Terracotta Eclipse plugin, you can locate and load modules directly from the Eclipse interface. Any required changes to `tc-config.xml` are written automatically. No manual editing of `tc-config.xml` is necessary.

For example, to cluster an instance of `org.apache.commons.collections.FastHashMap`, use the module `clustered-commons-collections-3.1-1.0.0-SNAPSHOT.jar` by modifying `tc-config.xml`:

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
  <!-- ...system and server stuff... -->
  <clients>
    <modules>
      <module name="clustered-commons-collections-3.1" version="1.0.0-SNAPSHOT"/>
    </modules>
  </clients>
  <!-- ...application stuff... -->
</tc:tc-config>
```

There is no limit to the number of modules as you can specify in `tc-config.xml`. You can also avoid specifying any modules and omit the `<modules>` section altogether.

Specifying Module Repositories

Terracotta module repositories can be specified similar to the way Maven 2 plugin repositories are specified. When a module is specified in `tc-config.xml`, the Terracotta runtime searches each module repository until it finds a filename match. Repositories are searched in the order listed in `tc-config.xml`, with the exception of `<terracotta-install-directory>/modules`, which is always searched first.

You can specify any number of other module repositories by using a valid file path (relative or absolute). For example:

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
  <!-- ...system and server stuff... -->
  <clients>
    <modules>
      <repository>/usr/local/share/terracotta/modules</repository>
      <!-- modules listed below -->
    </modules>
  </clients>
  <!-- ...application stuff... -->
</tc:tc-config>
```

For any module you specify, the Terracotta runtime would search the following directories in the order shown:

1. `<terracotta-install-directory>/modules`
2. Repositories specified in `tc-config.xml`.

The Terracotta runtime stops its search on finding the first occurrence of the file. In the example above, if the search fails to find the target file in the default Terracotta modules directory, it continues with `/usr/local/share/terracotta/modules`.

The Terracotta runtime does not check Maven repositories when running from the kit.

If you are running in Maven with the Terracotta Maven plugin, the Terracotta runtime searches *only* the local maven repository.

The group-id Attribute

In addition to the `name` and `version` attributes, you should specify a `group-id` if the TIM is not stored in the default Terracotta modules directory. The value of `group-id` should equal the domain-name portion of the package name (the domain name in reverse order). For example, if myCompany.com creates a TIM with the filename `myApp-2.1.0-1.0.0.jar`, and stores it in a new repository, then `tc-config.xml` could be configured this way:

```
<tc:tc-config xmlns:tc="http://www.terracotta.org/config">
  <!-- ...system and server stuff... -->
  <clients>
    <modules>
      <repository>/usr/local/share/terracotta/modules/myApp-2.1.0/1.1.0/com/myCompany/myApp</repository>
      <module name="myApp-2.1.0" version="1.1.0" group-id="com.myCompany.myApp"/>
    </modules>
  </clients>
  <!-- ...application stuff... -->
</tc:tc-config>
```

The path given as the value of the <repository/> element follows the convention of using the domain-name portion of the package name.

Configuration Variables

There are a few variables that will be interpolated by the configuration subsystem:

Variable	Interpolated Value
%h	The hostname
%i	The ip address
%D	Time stamp (yyyyMMddHHmmssSSS)
%(*system property*)	The value of the given system property

These variables will be interpolated in the following places:

- the "name", "host" and "bind" attributes of the <server> element
- the password file location for JMX authentication
- client logs location
- server logs location
- server data location

Value of %i

The variable %i is expanded into a value determined by the host's networking setup. In many cases that setup is in a `hosts` file containing mappings that may influence the value of %i.

Overriding tc.properties

Every Terracotta installation has a default `tc.properties` file containing system properties. Normally, the settings in `tc.properties` are pre-tuned and should not be edited.

If tuning is required, you can override certain properties in `tc.properties` using `tc-config.xml`. This can make a production environment more efficient by allowing system properties to be pushed out to clients with `tc-config.xml`. Those system properties would normally have to be configured separately on each client.

Setting System Properties in tc-config

To set a system property with the same value for all clients, you can add it to the Terracotta server's `tc-config.xml` file using a configuration element with the following format:

```
<property name="<tc_system_property>" value="<new_value>" />
```

All <property /> tags must be wrapped in a <tc-properties> section placed at the beginning of `tc-config.xml`.

For example, to override the values of the system properties `l1.cachemanager.enabled` and `l1.cachemanager.leastCount`, add the following to the beginning of `tc-config.xml`:

```
<tc-properties>
     <property name="l1.cachemanager.enabled" value="false" />
     <property name="l1.cachemanager.leastCount" value="4" />
</tc-properties>
```

Override Priority

System properties configured in `tc-config.xml` override the system properties in the default `tc.properties` file provided with the Terracotta kit. The default `tc.properties` file should **not** be edited or moved.

If you create a *local* `tc.properties` file in the Terracotta `lib` directory, system properties set in that file are used by Terracotta and will override system properties in the *default* `tc.properties` file. System properties in the local `tc.properties` file are **not** overridden by system properties configured in `tc-config.xml`.

System property values passed to Java using -D override all other configured values for that system property. In the example above, if `-Dcom.tc.l1.cachemanager.leastcount=5` was passed at the command line or through a script, it would override the value in `tc-config.xml` and `tc.properties`. The order of precedence is shown in the following list, with highest precedence shown last:

1. default `tc.properties`
2. `tc-config.xml`
3. local, or user-created `tc.properties` in Terracotta `lib` directory
4. Java system properties set with -D

Failure to Override

If system properties set in `tc-config.xml` fail to override default system properties, a warning is logged to the Terracotta logs. The warning has the following format:

```
The property <system_property_name> was set by local settings to <value>.
This value will not be overridden to <value> from the tc-config.xml file.
```

System properties used early in the Terracotta initialization process may fail to be overridden. If this happens, a warning is logged to the Terracotta logs. The warning has the following format:

```
The property <system_property_name> was read before initialization completed.
```

The warning is followed by the value assigned to <system_property_name>.

tc.properties Properties that Cannot be Overridden

`tc.management.mbeans.enabled` is known to load before initialization completes and cannot be overridden.

System Configuration Section

/tc:tc-config/system/configuration-model

The configuration-model element is for informational purposes. The two configuration-model options are **'development'** and **'production'**. These values have no effect on the functioning of Terracotta servers or clients, but instead allow you to designate the intended use of a configuration file using one of two recommended modes.

In development, you may want each client might have its own configuration, independent of the server or any other client. This approach is useful for development, but should not be used in production as it can result in shared data being corrupted or lost if, for instance, two different clients specify roots of different types. To note that a configuration file is intended for direct use by a client, set the configuration-model to **'development'**.

In production, each client should obtain its configuration from a Terracotta server instance. To note that a configuration file is intended be be fetched from a server, set the configuration-model to **'production'**.

In general, a client can specify that its configuration come from a server by setting the *tc.config* system propery:

```
-Dtc.config=serverHost:dsoPort
```

The default configuration model is **'development'**.

Sample configuration snippet:

```
   <configuration-model>production</configuration-model>
```

Servers Configuration Section

/tc:tc-config/servers

This section defines the Terracotta server instances present in your cluster. One or more entries can be defined. If this section is omitted, Terracotta configuration behaves as if there's a single server instance with default values.

Each Terracotta server instance needs to know which configuration it should use as it starts up. If the server's configured name is the same as the hostname of the host it runs on and no host contains more than one server instance, then configuration is found automatically.

If a configured Terracotta server instance has a name different from its host's name, then you must pass the command-line option `-n <name>` to the start-tc-server script, where <name> equals the name of the server instance as set in the configuration file. If you are running only one server instance, the command-line option does not need to be passed.

/tc:tc-config/servers/server

A server stanza encapsulates the configuration for a Terracotta server instance. The server element takes three optional attributes (see table below).

Attribute	Definition	Value	Default Value
host	The ID of the machine hosting the Terracotta server	Host machine's IP address or resolvable hostname	Host machine's IP address
name	The ID of the Terracotta server; can be passed to Terracotta scripts such as start-tc-server using -n <name>	user-defined string	<host>:<dso-port>
bind	The network interface on which the Terracotta server listens for Terracotta clients; 0.0.0.0 specifies all interfaces	interface's IP address	0.0.0.0

Sample configuration snippet:

```
<server>
  <!-- my host is '%i', my name is '%i:dso-port', my bind is 0.0.0.0 -->
  ...
</server>
<server host="myhostname">
  <!-- my host is 'myhostname', my name is 'myhostname:dso-port', my bind is 0.0.0.0 -->
  ...
</server>
<server host="myotherhostname" name="server1" bind="192.168.1.27">
  <!-- my host is 'myotherhostname', my name is 'server1', my bind is 192.168.1.27 -->
  ...
</server>
```

/tc:tc-config/servers/server/authentication

You can configure security using the Lightweight Directory Access Protocol (LDAP) or JMX authentication. Enabling one of these methods causes a Terracotta server to require credentials before allowing a JMX connection to proceed.

For more information on how to configure authentication, see the Terracotta Product Documentation.

/tc:tc-config/servers/server/data

This section lets you declare where the server should store its data.

Sample configuration snippet:

```
<!-- Where should the server store its persistent data? (This includes
   stored object data for DSO.) This value undergoes parameter substitution
   before being used; this allows you to use placeholders like '%h' (for the hostname)
   or '%(com.mycompany.propname)' (to substitute in the value of
   Java system property 'com.mycompany.propname'). Thus, a value
   of 'server-data-%h' would expand to 'server-data-artichoke' if
   running on host 'artichoke'.

   If this is a relative path, then it is interpreted relative to
   the location of this file. It is thus recommended that you specify
   an absolute path here.

   Default: 'data'; this places the 'data' directory in the same
   directory as this config file.
-->
<data>/opt/terracotta/server-data</data>
```

/tc:tc-config/servers/server/logs

This section lets you declare where the server should write its logs.

Sample configuration snippet:

```
<!-- In which directory should the server store its log files? Again,
   this value undergoes parameter substitution before being used;
   thus, a value like 'server-logs-%h' would expand to
   'server-logs-artichoke' if running on host 'artichoke'. See the
   Product Guide for more details.

   If this is a relative path, then it is interpreted relative to
   the location of this file. It is thus recommended that you specify
   an absolute path here.

   Default: 'logs'; this places the 'logs' directory in the same
   directory as this config file.
-->
<logs>/opt/terracotta/server-logs</logs>
```

You can also specify `stderr:` or `stdout:` as the output destination for log messages. For example:

```
<logs>stdout:</logs>
```

/tc:tc-config/servers/server/statistics

This section lets you declare where the server should store buffered statistics data.

Sample configuration snippet:

```
    <!-- In which directory should the server store statistics
       data that is being buffered? Again, this value undergoes
       parameter substitution before being used; thus, a value
       like 'statistics-data-%h' would expand to 'statistics-data'
       if running on host 'artichoke'. See the Product Guide for
       more details.

       If this is a relative path, then it is interpreted relative to the
       current working directory of the server (that is, the directory
       you were in when you started server). It is thus recommended
       that you specify an absolute path here.

       Default: 'statistics'; this places the 'statistics' directory in the
       directory you were in when you invoked 'start-tc-server'.
    -->
    <statistics>/opt/terracotta/server-statistics</statistics>
```

/tc:tc-config/servers/server/dso-port

This section lets you set the port that the Terracotta server listens to. It is called 'dso-port' for historical reasons (there used to be a distinction between the Distributed Shared Object service and other services in the Terracotta server).

The default value of 'dso-port' is 9510.

Sample configuration snippet:

```
    <dso-port>9510</dso-port>
```

/tc:tc-config/servers/server/http-authentication/user-realm-file

Turn on authentication for the embedded Terracotta HTTP Server. This requires a properties file that contains the users and passwords that have access to the HTTP server.

The format of the properties file is:

```
username: password [,rolename ...]
```

The supported roles and protected sections are:

- `statistics` (for the statistics gatherer at `/statistics-gatherer`, more information in the Cluster Statistics Recorder Guide.)

Passwords may be clear text, obfuscated or checksummed. The class `com.mortbay.Util.Password` should be used to generate obfuscated passwords or password checksums.

By default, HTTP authentication is turned off.

Sample configuration snippet:

```
<http-authentication>
    <user-realm-file>/opt/terracotta/realm.properties</user-realm-file>
</http-authentication>
```

/tc:tc-config/servers/server/jmx-port

This section lets you set the port that the Terracotta server's JMX Connector listens to.

The default value of 'jmx-port' is 9520.

Sample configuration snippet:

```
    <jmx-port>9520</jmx-port>
```

/tc:tc-config/servers/server/l2-group-port

This section lets you set the port that the Terracotta server uses to communicate with other Terracotta servers when the servers are run in network-based active-passive mode.

The default value of 'l2-group-port' is 9530.

Sample configuration snippet:

```
<l2-group-port>9530</l2-group-port>
```

/tc:tc-config/servers/server/dso

This section contains configuration specific to the Distributed Shared Object service of the Terracotta server.

/tc:tc-config/servers/server/dso/client-reconnect-window

This section lets you declare the client reconnect-time window. The value is specified in seconds and the default is 120 seconds. If adjusting the value in <c`lient-reconnect-window>`, note that a too-short reconnection window can lead to unsuccessful reconnections during failure recovery, and a too-long window lowers the efficiency of the network.

Sample configuration snippet:

```
      <client-reconnect-window>120</client-reconnect-window>
```

Further reading:
For more information on client and server reconnection is executed by Terracotta DSO, see the Concept and Architecture Guide.
For more information on tuning reconnection properties in a high-availability environment, see *Configuring and Testing Terracotta For High Availability* in .

/tc:tc-config/servers/server/dso/persistence

This section lets you configure whether Terracotta operates in persistent (permanent-store) or non-persistent mode (temporary-swap-only).

Permanent-store mode backs up all shared in-memory data to disk. This backed-up data survives server restarts and cluster failures, allowing all application state to be restored. This mode is recommended for ensuring a more robust failover architecture.

Temporary-swap-only mode is the default mode. This mode uses the disk as a temporary backing store. Data is not preserved across server restarts and cluster failures, although shared in-memory data may survive if a backup Terracotta server is set up.

Further reading:
For more information on Terracotta cluster persistence and restarting servers, see the architecture section of the Concept and Architecture Guide

Sample configuration snippet:

```
      <persistence>
        <!--
           Default: 'temporary-swap-only'
        -->
        <mode>permanent-store</mode>
      </persistence>
```

/tc:tc-config/servers/server/dso/garbage-collection

This section lets you configure the periodic distributed garbage collector (DGC) that runs in the Terracotta server.

Further reading:
For more information, see the distributed garbage collection section of the Concept and Architecture Guide

Configuration snippet:

```
       <garbage-collection>


         <!-- If 'true', distributed garbage collection is enabled.
            You should only set this to 'false' if you are
            absolutely certain that none of the data underneath
            your roots will ever become garbage; certain
            applications, such as those that read a large amount
            of data into a Map and never remove it (merely look up
            values in it) can safely do this.

            Default: true
         -->
         <enabled>true</enabled>

         <!-- If 'true', the DSO server will emit extra information when
            it performs distributed garbage collection; this can
            be useful when trying to performance-tune your
            application.

            Default: false
         -->
         <verbose>false</verbose>

         <!-- How often should the DSO server perform distributed
            garbage collection, in seconds?

            Default: 3600 (60 minutes)
         -->
         <interval>3600</interval>
       </garbage-collection>
```

/tc:tc-config/servers/ha

This section allows you to indicate properties associated with running your servers in active-passive mode. The properties apply to all servers defined. You can omit this section, in which case your servers, running in persistent mode, will run in networked active-passive mode.

In order to allow for at most 1 <ha> section to be defined along with multiple <server> sections, they must be defined in a given order (i.e., multiple <server> section then 0 or 1 <ha> section).

Sample configuration snippet:

```
<servers>
    <server host="%i" name="sample1">
    </server>
    <server host="%i" name="sample2">
    </server>
    <ha>
      <mode>networked-active-passive</mode>
    </ha>
</servers>
```

/tc:tc-config/servers/ha/mode

This section allows you configure whether servers run in network-based (default) or disk-based active-passive High Availability (HA) mode. Network-based servers have separate databases and sync over the network. Disk-based servers share the database, which is locked by the active server.

There are two corresponding value options: 'networked-active-passive' and 'disk-based-active-passive'.

When using networked-active-passive mode, Terracotta server instances must not share data directories. Each server's <data> element should point to a different and preferably local data directory.

Sample configuration snippet:

```
<mode>networked-active-passive</mode>
```

/tc:tc-config/servers/ha/networked-active-passive

This section allows you to declare the election time window, which is used when servers run in network-based active-passive mode. An active server is elected from the servers that cast a vote within this window. The value is specified in seconds and the default is 5 seconds. Network latency and work load of the servers should be taken into consideration when choosing an appropriate window.

Sample configuration snippet:

```
<networked-active-passive>
  <election-time>5</election-time>
</networked-active-passive>
```

/tc:tc-config/servers/mirror-groups

Use the <mirror-groups> section to bind groups of Terracotta server instances into a server array. The server array is built from sets of Terracotta server instances with one active server instance and one or more "hot standbys" (back-ups).

The following table summarizes the elements contained in a <mirror-groups> section.

Element	Definition	Attributes
<mirror-groups>	Encapsulates any number of <mirror-group> sections. Only one <mirror-groups> section can exist in a Terracotta configuration file.	None
<mirror-group>	Encapsulates one <members> element and one <ha> element.	`group-name` can be assigned a unique non-empty string value. If not set, Terracotta automatically creates a unique name for the mirror group.
<members>	Encapsulates <member> elements. Only one <members> section can exist in a <mirror-group> section.	None
<ha>	Applies high-availability settings to the encapsulating mirror group only. Settings are the same as those available in the main high-availability section. Mirror groups without an <ha> section take their high-availability settings from the main high-availability section.	None
<member>	Contains the value of the server-instance name attribute configured in a <server> element. Each <member> element represents a server instance assigned to the mirror group.	None

For examples and more information, see *Terracotta Server Arrays* in .

Clients Configuration Section

The clients section contains configuration about how clients should behave.

/tc:tc-config/clients/modules

The modules section allows you to include additional configuration modules. See Configuration Modules for details.

/tc:tc-config/clients/logs

This section lets you configure where the Terracotta client writes its logs.

Sample configuration snippet:

```
<!--
   This value undergoes parameter substitution before being used;
   thus, a value like 'client-logs-%h' would expand to
   'client-logs-banana' if running on host 'banana'. See the
   Product Guide for more details.

   If this is a relative path, then it is interpreted relative to
   the current working directory of the client (that is, the directory
   you were in when you started the program that uses Terracotta
   services). It is thus recommended that you specify an absolute
   path here.


   Default: 'logs-%i'; this places the logs in a directory relative
   to the directory you were in when you invoked the program that uses
   Terracotta services (your client), and calls that directory, for example,
   'logs-10.0.0.57' if the machine that the client is on has assigned IP
   address 10.0.0.57.
-->
<logs>logs-%i</logs>
```

You can also specify `stderr:` or `stdout:` as the output destination for log messages. For example:

```
<logs>stdout:</logs>
```

/tc:tc-config/clients/statistics

Beginning with Terracotta 3.3.0, this element has been removed.

/tc:tc-config/clients/dso/fault-count

Sets the object fault count. Fault count is the maximum number of reachable objects that are pre-fetched from the Terracotta server to the Terracotta client when an object is faulted from that server to that client.

Pre-fetching an object does not fault the entire object, only the minimum amount of metadata needed to construct the object on the client if necessary. Unused pre-fetched objects are eventually cleared from the client heap.

Objects qualify for pre-fetching either by being referenced by the original object (being part of original object's object graph) or because they are peers. An example of peers are primitives found in the same map.

In most applications, pre-fetching improves locality of reference, which benefits performance. But in some applications, for example a queue fed by producer nodes that shouldn't pre-fetch objects, it may be advantageous to set the fault count to 0.

Sample configuration snippet:

```
      <fault-count>500</fault-count>
```

Default: *500*

/tc:tc-config/clients/dso/debugging

The client debugging options give you control over various logging output options to gain more visibility into what Terracotta is doing at runtime.

/tc:tc-config/clients/dso/debugging/instrumentation-logging

Instrumentation logging provides logging output about various instrumentation events.

/tc:tc-config/clients/dso/debugging/instrumentation-logging/class

If set to 'true', this option will log when a class is loaded that matches an include pattern. You can use this to find out which classes are actually being instrumented and which classes are not.
Default: ***false***
See also: configuring inclusion and exclusion of classes for instrumentation
See also: more on class instrumentation in the Concept and Architecture Guide

/tc:tc-config/clients/dso/debugging/instrumentation-logging/roots

If set to 'true', this option will log when a DSO root object is instrumented. You can use this to find out which roots are being included in the application.
Default: ***false***
See also: configuring roots
See also: more on roots in the Concept and Architecture Guide

/tc:tc-config/clients/dso/debugging/instrumentation-logging/locks

If set to 'true', this option will log when a method in a class matches a lock pattern. You can use this to find out which methods are actually configured for locking and which methods are not.
Default: ***false***
See also: configuring locks
See also: more on locks in the Concept and Architecture Guide

/tc:tc-config/clients/dso/debugging/instrumentation-logging/transient-root

If set to 'true', this option will log when a field that has been declared as a root is also transient (either by being flagged with the 'transient' keyword or declared as transient in the Terracotta configuration). The field will be a root, it will not be transient
Default: ***true***
See also: configuring transient fields
See also: more on transience in the Concept and Architecture Guide

/tc:tc-config/clients/dso/debugging/instrumentation-logging/distributed-methods

If set to 'true', this option will log when a method in a class is a distributed method.
Default: ***false***
See also: configuring Distributed Method Invocation
See also: the Distributed Method Invocation section of the Concept and Architecture Guide

/tc:tc-config/clients/dso/debugging/runtime-logging

Runtime logging provides logging output about various runtime operations on clustered objects and locks.

/tc:tc-config/clients/dso/debugging/runtime-logging/non-portable-dump

When a non-portable object is encountered in the graph of a newly shared object instance, the object graph of that newly shared instance is dumped to the terracotta log file. So for example, if you were attempting to share an instance of some class com.example.Portable, which in turn contained a LinkedList containing a non-portable object, you'd get a TCNonportableObjectError that says that LinkedList contains a reference to the non-portable type. The object dumper would log the entire graph of the com.example.Portable instance in the log file.

Here's some sample output that one would see in the log file (lines that start with !! are non-portable types):

```
!! com.tctest.NonPortableInstancesTest$NotPortable (id 0)
     Map m = (HashMap, id 1)
       [entry 0]
         key = "ref"
         value = (com.tctest.NonPortableInstancesTest$Ref, id 2)
           Object ref = (com.tctest.NonPortableInstancesTest$Ref, id 4)
!!           Object ref = (ref id 0)
!!   Thread t = (java.lang.Thread) (never portable)
```

Some notes about the above output:

- All objects include an "id", such that circular references can be traced. In this case, there is a circular reference back to the root object containing the output "(ref id 0)".
- The qualifier "(never portable)" in the last line indicates a type that is never allowed in a DSO object graph.
- Type names in java.lang.* and java.util.* are trimmed, resulting in names like "HashMap" and "Thread" instead of long package names.

Here's is some more complicated output (showing arrays and collections, etc.):

```
com.tc.object.walker.BasicWalkerTest$Root (id 0)
  Object b = (com.tc.object.walker.BasicWalkerTest$B, id 1)
    int com.tc.object.walker.BasicWalkerTest$A.i = 0
    int com.tc.object.walker.BasicWalkerTest$B.i = 1
  Collection c = (ArrayList, id 2)
    [0] = (com.tc.object.walker.BasicWalkerTest$Foo, id 3)
      int i = 2
      com.tc.object.walker.BasicWalkerTest$Foo next = (id 4)
        int i = 1
        com.tc.object.walker.BasicWalkerTest$Foo next = null
    [1] = (TreeMap, id 5)
      [entry 0]
        key = "k"
        value = null
      [entry 1]
        key = "key"
        value = (com.tc.object.walker.BasicWalkerTest$Foo, id 6)
          int i = 3
          com.tc.object.walker.BasicWalkerTest$Foo next = null
    [2] = (ref id 2)
    [3] = null
    [4] = 3.141592653589793
  Class clazz = class com.tc.object.walker.BasicWalkerTest$Root
  double[] emptyArray = (id 7)
  LinkedList emptyList = (id 8)
  HashMap emptyMap = (id 9)
  int i = 12
  int[] intArray = (id 10)
    [0] = 1
    [1] = 2
    [2] = 3
  Map m = (LinkedHashMap, id 11)
    [entry 0]
      key = "timmy"
      value = (com.tc.object.walker.BasicWalkerTest$Foo, id 12)
        int i = 4
        com.tc.object.walker.BasicWalkerTest$Foo next = null
    [entry 1]
      key = "yo"
      value = null
    [entry 2]
      key = "foo"
      value = (com.tc.object.walker.BasicWalkerTest$Foo, id 13)
        int i = 5
        com.tc.object.walker.BasicWalkerTest$Foo next = null
    [entry 3]
      key = "foo foo"
      value = (com.tc.object.walker.BasicWalkerTest$Foo, id 14)
        int i = 7
        com.tc.object.walker.BasicWalkerTest$Foo next = (id 15)
          int i = 6
          com.tc.object.walker.BasicWalkerTest$Foo next = null
  Object[][] multi = (id 16)
    [0] = (Object[], id 17)
      [0] = "timmy"
      [1] = 4
    [1] = (Object[], id 18)
      [0] = null
      [1] = null
    [2] = (Object[], id 19)
      [0] = null
      [1] = (ref id 0)
  Object[] objArray = (id 20)
    [0] = (ref id 0)
    [1] = (com.tc.object.walker.BasicWalkerTest$Foo, id 21)
      int i = 0
      com.tc.object.walker.BasicWalkerTest$Foo next = null
    [2] = 3
  String s = "root"
  Object self = (ref id 0)
```

Default: ***true***
See also: the Portability section of the Concept and Architecture Guide

/tc:tc-config/clients/dso/debugging/runtime-logging/lock-debug

When set to true, every lock acquisition will be logged. Lock releases are not logged.
Default: ***false***

/tc:tc-config/clients/dso/debugging/runtime-logging/wait-notify-debug

When set to true, every clustered wait and notify call is logged.
Default: ***false***

/tc:tc-config/clients/dso/debugging/runtime-logging/distributed-method-debug

When set to true, every distributed method call is logged.
Default: ***false***

/tc:tc-config/clients/dso/debugging/runtime-logging/new-object-debug

When set to true, the addition of every new clustered object is logged.
Default: ***false***

/tc:tc-config/clients/dso/debugging/runtime-logging/named-loader-debug

When set to true, a log entry similar to the following is added each time a named classloader is registered with Terracotta:

```
2009-01-16 12:59:23,973 [main] INFO com.terracottatech.dso.runtime -
   loader of type [sun.misc.Launcher$AppClassLoader] with name [Standard.system] registered (replaced: false)
```

The logged message shows the runtime type of the classloader, the classloader name, and a Boolean value indicating whether a previous registration for the same name is being replaced.

/tc:tc-config/clients/dso/debugging/runtime-output-options

The runtime output options are modifiers to the runtime logging options

/tc:tc-config/clients/dso/debugging/runtime-output-options/auto-lock-details

When set to true, this option provides more verbose output.
Default: ***false***

/tc:tc-config/clients/dso/debugging/runtime-output-options/caller

When set to true, this option logs the call site in addition to the runtime output.
Default: ***false***

/tc:tc-config/clients/dso/debugging/runtime-output-options/full-stack

When set to true, this option provides a full stack trace of the call site in addition to the runtime output.
Default: ***false***

Application Configuration Section

/tc:tc-config/application/dso/instrumented-classes

The instrumented-classes section determines which classes should be instrumented—i.e., have clustering capabilities injected into them—by Terracotta. The class of every object that is to become a clustered object as well as any class that manipulates clustered objects or acquires clustered locks must be instrumented. It's possible to declare that every class be instrumented; this is the safest and easiest approach when starting with Terracotta. However, restricting the set of instrumented classes to only those that must be instrumented will shorten startup time (because there is less code weaving to do as classes are loaded) and will improve performance at runtime. As you become more familiar with the set of classes that your application actually requires to be instrumented, we recommend that you tune your includes statements to this more restricted set of classes.

The instrumented-classes section allows both "include" and "exclude" stanzas. Each include or exclude stanza uses an AspectWerkz Class selection expression which is a regular expression that describes a set of classes.

When running on Java 5, Class Selector can select classes based on arbitrary annotations.

The include and exclude stanzas are evaluated in order beginning with the bottom stanza and working up. The first matching pattern determines whether the class will or will not be instrumented. If no patterns match, the class will not be instrumented. Using a combination of include and exclude class expressions, you can pick out exactly the set of classes that you want included for instrumentation.

In addition to selecting a set of classes for inclusion, the "include" stanza also provides further configuration options for how instances of classes picked out by that stanza are treated by Terracotta. Those additional configuration options are "honor-transient" and "on-load."

The "honor-transient" option, if set to "true," tells Terracotta to honor the transience of a field in the class as declared by the "transient" keyword. This will direct Terracotta to ignore fields marked as transient and not include those fields in the shared object graph. This is similar to how the transient keyword directs the Java serialization mechanism to leave transient fields out of the serialized object stream.

The "on-load" option allows you to specify a mechanism for initializing an instance of a class picked out by the include stanza when it is loaded (a.k.a. 'hydrated', 'materialized', or 'instantiated') into a JVM by Terracotta. You can choose a method to be called on that class or write an inline BeanShell script.

For technical reasons related to the implementation of BeanShell, you must refer to the current object as 'self' rather than 'this' in a BeanShell script.

Default: *false*
Further reading:
More on the AspectWerkz Pattern Language.
More on what can and can't be clustered with Terracotta
More on transient fields in Terracotta

Sample configuration snippet:

```
<instrumented-classes>
  <!-- This includes a certain set of classes for instrumentation. -->

  <include>
    <!-- The class(es) to include, as an AspectWerkz-compatible
       class specification expression. See the Product Guide
       for more details. (REQUIRED) -->
    <class-expression>com.mycompany.pkga.*</class-expression>

    <!-- If set to 'true', then any fields in the given class(es)
       that have the Java 'transient' modifier are not shared
       across JVMs. If set to 'false', they are.

       Default: false -->
    <honor-transient>true</honor-transient>
    <on-load>
      <!--This method will be called on instances of the specified class
          on load of the object. Used to rewire transients. -->
            <!--<method>aMethod</method>-->

      <!-- This bean shell script is called right after an object is loaded -->
            <execute><![CDATA[self.myTransientField = new ArrayList();]]></execute>
       </on-load>
  </include>

  <!-- The class(es) to exclude, as an AspectWerkz-compatible
     class specification expression. See the Product Guide for
     more details. -->
  <exclude>com.mycompany.pkga.subpkg.*</exclude>

  <include>
    <class-expression>com.mycompany.pkgb.*</class-expression>
  </include>
</instrumented-classes>
```

For inner classes, use a $ separator:

```
<class-expression>my.package.MyOuterClass$MyInnerClass</class-expression>
```

A * character can also be used to match inner classes:

```
<class-expression>my.package.MyOuterClass*</class-expression>
```

Also see this gotcha.

/tc:tc-config/application/dso/transient-fields

This section allows you to specify specific fields as transient to Terracotta.

Currently Terracotta doesn't support selection of the transient fields based on Java 5 annotations.

When a field should be made transient to Terracotta but hasn't been declared transient in the source code, and modifying the source code is not feasible, declare the fields transient in this section.

Example configuration snippet:

```
<transient-fields>
  <field-name>com.mycompany.pkga.MyClassOne.fieldA</field-name>
  <field-name>com.mycompany.pkgb.subpkg2.fieldB</field-name>
</transient-fields>
```

Further Reading:
More on transient fields in Terracotta

/tc:tc-config/application/dso/locks

The locks configuration section allows you to specify the Terracotta locking semantics. Each configuration stanza picks out a set of methods using the AspectWerkz Method selection expressions, similar to the way the instrumented-classes stanzas pick out sets of classes.

When running on Java 5, method selector can select methods based on annotations.

Each stanza determines how the set of methods it picks out are locked. There are two styles of locking: named locks and "autolocks."

Lock Style

Named locks require the thread executing a method to acquire a clustered lock of the specified name. This is a very coarse-grained locking style and is usually not appropriate for production code. However, it is useful when you are just getting started with Terracotta, or you must lock a method that doesn't have synchronization in the source code and it isn't feasible to change that source code.

Autolocks transform any synchronization (including wait() and notify() calls) on clustered objects within the scope of a method into cluster-wide synchronization on those objects. For this style of locking to be useful, there must be synchronization code in that method AND that synchronization must be performed on clustered objects. If the original definition of the method is not synchronized, one could enable the "auto-synchronized" attribute. When set to true, this attribute will add a synchronized modifier to the method signature before autolocking on the method. By default, "auto-synchronized" attribute is set to false.

To summarize, the following rules apply to autolocking:

1. Methods to be autlocked must have synchronization. Autolocking a method that has no synchronization does nothing.
2. The method must be configured (in the Terracotta configuration) to autolock.
3. The object synchronized on must be clustered.
4. The class with the locking in it must be included (in Terracotta configuration).

Lock Level

There are four lock levels: '**write**', '**synchronous-write**', '**read**', and '**concurrent**'

If unspecified, the default lock level is '**write**'.

Write locks are used when the method(s) being locked modify the data being protected. Only one thread in one JVM, cluster-wide, may be inside the same write lock at once. Further, no threads may be inside a read lock while one thread holds a write lock of the same identity.

Synchronous-write locks add a further guarantee over write locks. Where a write lock guarantees that all changes made in the scope of that lock will be applied prior to a thread acquiring that lock, synchronous-write locks guarantee that the thread holding the lock will not release the lock until the changes made under that lock are fully ACKed by the server.

Read locks may be used only when the method(s) being locked do not actually modify the data in question; multiple threads in multiple JVMs may be inside the same read lock at the same time. However, if a thread is locked upon the same object but using a write lock, then any threads that require a read lock must wait until the write locks has been released.

Concurrent locks allow multiple threads to be inside a concurrent lock at the same time, and all can write to the data. This can lead to nondeterministic program behavior, thus requiring considerable caution.

CONCURRENT LOCKS ARE FOR ADVANCED USE ONLY AND STRONGLY DISCOURAGED. THIS FEATURE MAY BE DISCONTINUED WITHOUT WARNING.

Further reading:
More on locks in Terracotta

Sample configuration snippet:

```
<locks>
 <!-- Specifies an autolock. -->
 <autolock auto-synchronized="false">
   <!-- The expression of method(s) to lock. This is an
      AspectWerkz-compatible method specification expression.
      (REQUIRED) -->
   <method-expression>* com.mycompany.pkga.MyClassOne.set*(..)</method-expression>

 <!-- The level of the lock: 'write', 'synchronous-write', 'read', or
      'concurrent'.
   <lock-level>write</lock-level>
 </autolock>

 <!-- Specifies a named lock. If you create multiple named-lock
    sections in this file that share the same name, they are
    the same lock, and will behave as such. -->
 <named-lock>
   <!-- The name of the lock. (REQUIRED) -->
   <lock-name>lockOne</lock-name>

   <!-- The expression of method(s) to lock. This is an
      AspectWerkz-compatible method specification expression.
      (REQUIRED) -->
   <method-expression>* com.mycompany.pkgb.get*(int)</method-expression>

   <!-- The level of the lock: 'read', 'write', or 'concurrent'.
      See above for more details.
   -->
   <lock-level>read</lock-level>
 </named-lock>
</locks>
```

/tc:tc-config/application/dso/roots

The "roots" section of the Terracotta configuration allows you to declare roots by name and bind them to specific fields in classes. Each root stanza requires either an explicit field name (any field in any class) or a field expression. If you don't specify a root name, the root name will be implied by the qualified name of the java class field that the root is bound to.

When running on Java 5, the field expression can select root fields based on annotations

A root may be assigned to more than one field by declaring multiple root stanzas in the configuration. Each stanza would have the same root name, but bind that root to different fields.

A field that is declared a root is endowed with special properties by Terracotta. If the root specified by the given root name has not been created yet, the first assignment to that root field will cause the root to be created. If the root specified by the given root name has already been created (at any other time by any thread in any virtual machine), all assignments to that root field will be ignored and replaced with an assignment to the existing root object.

This usually implies a small code change around root field setting. Generally, this means that you should check to see if the root has already been created and initialized prior to running any initialization code for that root object.

Further reading:

- More on roots in the Concept and Architecture Guide
- Logging transient roots

Sample configuration snippet:

```
<roots>
       <root>
         <!-- One of either field-name or field-expression is REQUIRED,
              but not BOTH -->

         <!-- The name of the field to make a root. This must be
            the fully-qualified name of a field. -->
         <field-name>com.mycompany.pkgc.AnotherClass.field1</field-name>

         <!-- A field expression to select fields to be roots. This must be
              a valid AspectWerkz-compatible field expresion.-->
         <!-- <field-expression>@com.example.MyRootAnnotation * *</field-expression> -->

         <!-- The name for this root. This is optional, but can
            help when debugging your system. -->
         <root-name>rootOne</root-name>
       </root>

       <root>
         <field-name>com.mycompany.pkgc.ClassTwo.field3</field-name>
         <root-name>rootTwo</root-name>
       </root>
     </roots>
```

/tc:tc-config/application/dso/app-groups

Sharing Terracotta roots allows different applications to share application state. The benefits from sharing roots include allowing a remote monitoring application to view the current shared state of multiple applications and providing a seamless application state to user sessions interacting with two or more applications.

Use the <app-groups> section to allow different applications to effectively share roots without failing on classloader errors. These errors, which manifest as ClassCastException and ClassNotFoundException, can be caused when different clustered applications need to share object graphs but use different classloaders.

com.tc.loader.system.name Property Deprecated

With previous versions of Terracotta, you might have prevented classloader-related exceptions by setting the `com.tc.loader.system.name` property (`-Dcom.tc.loader.system.name`) to name classloaders. However, this property has been deprecated and support for it may be dropped. It is recommended that you upgrade to Terracotta 3.0 or above and begin using app-groups.

The elements composing the <app-groups> section are listed in the following table.

Element	Definition	Attributes
<app-groups>	Encapsulates any number of <app-group> sections, one for each set of applications that need to share roots. Only one <app-groups> section can exist in a Terracotta configuration file.	None
<app-group>	Encapsulates any number of <web-application> and <named-classloader> elements.	`name` (required) must be assigned a unique non-empty string value.
<web-application>	Contains the context of the web application being clustered.	None
<named-classloader>	Contains the classloader used by the application being clustered. You may be able to find a named classloader using the Terracotta logs (see #named-loader-debug).	None

Restrictions

Certain restrictions apply to using the app-groups section:

- Applications in a single application group must be running in different JVMs.
- Applications in a single application group must be running in different web containers.
- The same container, and the same container version, must be used for all web applications configured in an <app-group> section.

Example: Web Applications Sharing Roots

AppA and AppB are running on one JVM, while AppC and AppD are running on another JVM. AppA and AppC need to share roots, and AppB and AppD need to share roots. The following configuration makes that sharing possible:

```
<application>
 <dso>
    <app-groups>
      <app-group name="ac">
        <web-application>AppA</web-application>
        <web-application>AppC</web-application>
      </app-group>
      <app-group name="bd">
        <web-application>AppB</web-application>
        <web-application>AppD</web-application>
      </app-group>
    </app-groups>
 </dso>
</application>
```

This example can be reduced to two web applications by having only one <app-group> section.

Example: Web Application and POJO Application Sharing Roots

Web application AppA is running on one JVM, while a POJO application is running on another JVM. The two applications need to share roots. The following configuration makes that sharing possible:

```
<application>
 <dso>
    <app-groups>
      <app-group name="webAndPojo">
        <web-application>AppA</web-application>
        <named-classloader>Standard.system</named-classloader>
      </app-group>
    </app-groups>
 </dso>
</application>
```

In this example, the POJO application's classloader must be named in a <named-classloaer> element. Unlike web applications, there's no option to use the application's context name in a <web-application> element.

/tc:tc-config/application/dso/injected-instances

Use the <injected-instances> section to add Terracotta functionality directly in your application. The functionality is injected into a specified field, while the field's type determines which instance is actually injected.

This section has the following elements:

- <injected-instances> – Encapsulates any number of <injected-field> sections.
- <injected-field> – Encapsulates one <field-name> element.
- <field-name> – Specifies the fully qualified name of the class being injected.
- <instance-type> – Specifies the injected instance type. Useful in cases where the target field's type is insufficient to determine the correct instance to inject.

Classes that are *not* configured for instrumentation by Terracotta cannot be injected using this configuration.

Example: Injecting Cluster Awareness

You can use the <injected-instances> section to inject instrumented classes for cluster awareness, giving them the ability to listen for cluster events. For example, to inject the classes `ClusterAwareClass` and `OtherClusterAwareClass`, add the following <injected-instance> subsection to the Terracotta configuration file's clients/dso section:

```
<clients>
 <dso>
  ...
   <injected-instances>
     <injected-field>
       <field-name>com.mypackage.myClasses.ClusterAwareClass.theField</field-name>
     </injected-field>
     <injected-field>
       <field-name>com.mypackage.myClasses.OtherClusterAwareClass.theOtherField</field-name>
     </injected-field>
   </injected-instances>
   ...
 </dso>
</clients>
```

In the code above, `ClusterAwareClass` and `OtherClusterAwareClass` are instrumented for injection. When an instance of either class is constructed or faulted onto a shared object graph, it will be injected with cluster awareness.

Both `ClusterAwareClass` and `OtherClusterAwareClass` must also be configured for Terracotta instrumentation (see #Application Configuration Section).

For more information on cluster events, see Cluster Events.

/tc:tc-config/application/dso/distributed-methods

The distributed-methods section of the Terracotta config allows you to specify which methods are eligible for Distributed Method Invocation (DMI). Each stanza picks out a set of methods using an AspectWerkz Method selection expression.

When running on Java 5, method selector can select methods based on annotations

The "run-on-all-nodes" attribute allows you determine whether or not to force the object on which a distributed method invocation is to be executed to be loaded and the method executed should the object not be present in all JVMs in the cluster. If "run-on-all-nodes" is false, the method will be executed only on those objects that happen to be loaded in client JVMs. If "run-on-all-nodes" is true, then, if the object isn't loaded yet in a particular client JVM, the object will automatically be loaded and the method executed on that object.

Further reading:
More on distributed method invocation
More on the AspectWerkz Pattern Language

Sample configuration snippet:

```
<!-- This section specifies methods to invoke in a distributed
   fashion via Terracotta DSO. When a method matching one of the
   patterns here is invoked in one JVM, Terracotta DSO will
   cause it to be invoked simultaneously in all JVMs throughout
   the Terracotta cluster. This is often used for 'event listener'
   methods (for example, those used by Swing), to make sure that
   all JVMs throughout the cluster are informed when a particular
   event occurs. -->
<distributed-methods>
  <!-- An AspectWerkz-compatible method specification expression
     denoting which method(s) to distribute. -->
  <method-expression>
    void com.mycompany.pkga.MyClass.somethingHappened(String, int)
  </method-expression>
  <method-expression>
    String[] com.mycompany.pkgc.AnotherClass.eventOccurred(Boolean, Double)
  </method-expression>
  <!-- An optional attribute run-on-all-nodes (default value "true")
     can be set to false to execute distributed only on those nodes
     that already have a reference to the object on which the method
     is called -->
  <method-expression run-on-all-nodes="false">
    String[] com.mycompany.pkgc.AnotherClass.eventOccurred(Boolean, Double)
  </method-expression>

</distributed-methods>
```

/tc:tc-config/application/dso/additional-boot-jar-classes

The "additional-boot-jar-classes" section lets you declare that additional classes other than the Terracotta default be placed into the Terracotta boot jar. When the boot jar is generated, Terracotta automatically adds certain classes to it by default. You can choose to add others as needed by declaring them in the "additional-boot-jar-classes" section and re-generating the Terracotta boot jar.

Further reading:
More on the Terracotta boot jar in the Concept and Architecture Guide

Sample configuration snippet:

```
<additional-boot-jar-classes>
  <include>java.awt.datatransfer.Clipboard</include>
</additional-boot-jar-classes>
```

/tc:tc-config/application/dso/web-applications

This section declares which web applications should use Terracotta session clustering. Each web application having clustered sessions must have an entry in this section.

Further reading:

- For standard (non-DSO) session clustering, see the .

Sample configuration snippet:

```
<web-applications>
  <web-application>petstore</web-application>
  <web-application>billing_app</web-application>
  <web-application>ROOT</web-application>
</web-applications>
```

For the default context path (ie. "/"), use the special value 'ROOT' (without the quotes).

Session Locking

By default, automatic session locking is in effect for web applications configured for clustered sessions. This can cause "starvation" or deadlock issues in certain applications when long-running requests block short-running requests. For example, AJAX-based applications are prone to these issues.

To prevent these issues, automatic session locking should be turned off using the <web-application> element's `session-locking` attribute:

```
 <web-applications>
   <web-application session-locking="false">myClusteredWebapp</web-application>
</web-applications>
```

With automatic session locking turned off, manual synchronization should be used for shared data.

If, however, a greater level of data security is required, session locking can be upgraded to synchronous write locks using the `synchronous-write` attribute:

```
<web-applications>
  <web-application synchronous-write="true">myClusteredWebapp</web-application>
</web-applications>
```


Performance Impact of Synchronous Write Locks

Enabling synchronous write locks can substantially degrade cluster performance because clients must wait for server acknowledgement before releasing a synchronous write lock.

Avoiding DSO Clustering of Session Objects

If the objects in your session are Serializable, you can use the <web-application> element's `serialization` attribute to prevent sharing (by DSO) of objects in the session:

```
<web-applications>
  <web-application serialization="true">myClusteredWebapp</web-application>
</web-applications>
```

In this example, the default value of the `session-locking` attribute is automatically switched to "false," which allows concurrent requests in the session. If single-request concurrency is a requirement, set `session-locking` explicitly:

```
<web-applications>
  <web-application serialization="true" session-locking="true">myClusteredWebapp</web-application>
</web-applications>
```

In serialization mode, sessions are still clustered, but your application must now follow the standard servlet convention on using setAttribute() for mutable objects in replicated sessions.

/tc:tc-config/application/dso/dso-reflection-enabled

This section lets you turn reflection sensitivity on and off. By default, Terracotta is sensitive to access to fields (including arrays) of clustered objects made through reflection. However, there is some overhead to this sensitivity. If your application heavily uses reflection to access fields, but never accesses clustered fields through reflection, you can turn off that sensitivity.

Sample configuration snippet:

```
<dso-reflection-enabled>false</dso-reflection-enabled>
```