Integrating Terracotta DSO

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 »**

Integrating Terracotta DSO

Introduction

Terracotta Distributed Shared Objects (DSO integrates with many of the most powerful Java technologies available today. This document includes resources and provides tips on integrating Terracotta DSO with popular frameworks and libraries.

To learn about integrating with Terracotta products, including Ehcache and Sessions, second-level cache for Hibernate, and Spring, see the

Error rendering macro 'html'

Notify your Confluence administrator that "Bob Swift Atlassian Add-ons - HTML" requires a valid license. Reason: EXPIRED

.

Integrating with a specific application server such as Tomcat is covered in

Error rendering macro 'html'

Notify your Confluence administrator that "Bob Swift Atlassian Add-ons - HTML" requires a valid license. Reason: EXPIRED

. For example, clustering a web application on Tomcat is covered in *Clustering Web Applications with Terracotta* in

Error rendering macro 'html'

Notify your Confluence administrator that "Bob Swift Atlassian Add-ons - HTML" requires a valid license. Reason: EXPIRED

.

How Integration Works

Integration with Terracotta DSO is implemented using Terracotta Integration Modules (TIMs), which are installed along with Terracotta clients on the application servers in a cluster. A TIM is a set of configuration elements packaged together as a single, includable module within the Terracotta configuration. Including a TIM in your project is as simple as adding one line to a configuration file.

In the following example, support for clustering Spring Security 2.0 with Terracotta DSO is achieved by adding the following line to the Terracotta configuration file:

```
<module name="tim-spring-security-2.0" />
```

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.

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.

TIMs are available for many software products. The easiest and most efficient way to download and install TIMs is through the tim-get.

You can view the full complement of TIMs in any of the following ways:

- See the Terracotta Forge
- Run `tim-get.sh list` to see top-level TIMs available to the current installation of Terracotta.
- Run `tim-get.sh list --all` to see all TIMs available to the current installation of Terracotta.

For the latest certified versions of platforms, including types and versions of JDKs, see the platforms page.

The Terracotta Toolkit

For certain TIMs, the Terracotta Toolkit is installed as a dependency. You must put the Terracotta Toolkit JAR, `terracotta-toolkit-<API version>-<version>` (open source) or `terracotta-toolkit-<API version>-ee-<version>` (Enterprise Edition), on your application's classpath.

Installing Both Open Source and Enterprise Edition TIMs

If you install both open-source TIMs and Enterprise Edition TIMs, then you must specify both types of Terracotta Toolkit JARs in the Terracotta configuration file. For example, if you want to install `tim-tomcat-6.0` and `tim-ehcache-2.x-ee`, then specify the following:

```
<modules>
  <module group-id="org.terracotta.toolkit" name="terracotta-toolkit-1.2" />
  <module group-id="org.terracotta.toolkit" name="terracotta-toolkit-1.2-ee" />
  <module name="tim-tomcat-6.0" />
  <module name="tim-ehcache-2.x-ee" />
<!-- Other TIMs here. -->
</modules>
```

The Terracotta Toolkit API version available for your Terracotta kit may be different than the one shown in this example.

Available Terracotta DSO Integrations

See the http://www.terracotta.org/Community for more news on Terracotta community members' success with integrating Terracotta.

Apache Struts

Versions Supported	1.1

Introduction

Apache Struts is a free open-source framework for creating Java web applications.

Dependencies

None.

Implementation

To enable clustering with Apache Struts 1.1, add the Struts configuration module to `tc-config.xml`:

```
<modules>
    <module name="tim-apache-struts-1.1" />
</modules>
```

Then run the following command from ${TERRACOTTA_HOME} to automatically install the correct version of the TIM:

UNIX/Linux

```
[PROMPT] bin/tim-get.sh install-for <path/to/tc-config.xml>
```

Microsoft Windows

```
[PROMPT] bin\tim-get.bat install-for <path\to\tc-config.xml>
```

Known Issues

None.

iBatis

Versions Supported	2.2.0

Introduction

The Apache iBatis Data Mapper framework is an ORM solution that makes it easier to use a database with Java and .NET applications.

iBatis allows lazy loading. When enabled, a POJO object returned from an iBatis query will contain a proxy object to the referenced object. For instance, a Customer object returned from a query to the Customer table will have a proxy object in its reference to the Account object. iBatis will de-reference the proxy object when the Account object is queried at a later point in time.

With the Terracotta integration module for iBatis, one node can query a Customer table lazily and share the returned Customer object, while a second node can de-reference the Account object reference and obtain the referenced Account object.

Dependencies

None.

Implementation

To enable clustering with iBatis, add the iBatis configuration module to `tc-config.xml`:

```
<modules>
    <module name="tim-iBatis-<iBatis_version>"/>
</modules>
```

For example, use the module `tim-iBatis-2.2.0` for clustering iBatis 2.2.0, add the following snippet to `tc-config.xml`:

```
<modules>
    <module name="tim-iBatis-2.2.0"/>
</modules>
```

Then run the following command from ${TERRACOTTA_HOME} to automatically install the correct version of the TIM:

UNIX/Linux

```
[PROMPT] bin/tim-get.sh install-for <path/to/tc-config.xml>
```

Microsoft Windows

```
[PROMPT] bin\tim-get.bat install-for <path\to\tc-config.xml>
```

Known Issues

None.

Lucene

Introduction

The Apache Lucene project develops open-source search software. Lucene support is included via the Compass and written about here.

The Terracotta integration module `tim-searchable`, available through the Terracotta Forge, is reported to improve clustering of large Lucene indexes.

Integrating Lucene with Terracotta

By Nam Chu

Overview

This section will show how to integrate Lucene 2.3.2 into Terracotta using the Compass TIM.

1. Download a Compass release. In this document, we use the Compass 2.0.2 release.

2. Unzip the downloaded file.

3. Go to the unzip folder - then go to the "dist" folder, you'll need these files to work:

- compass-2.0.2.jar (TIM)
- commons-logging.jar
- lucene-core.jar (which is the Lucene 2.3.2 release) in the "lucene" folder

Add these 3 jar files to our application classpath.

4. Next, create a folder structure such as TC_HOME/modules/org/compass-project/compass/2.0.2 and drop the compass-2.0.2.jar file in this directory and add the following to the tc-config file:

```
<modules>
      <module group-id="org.compass-project" name="compass" version="2.0.2"/>
 </modules>
```

Alternatively, if you do not want to create a separate folder, you can specify a repository in the tc-config file:

```
<modules>
      <!-- You may change the line below to point at the right folder -->
      <repository>/home/neo/workspace/TerracottaLucene/lib</repository>
      <module group-id="org.compass-project" name="compass" version="2.0.2"/>
 </modules>
```

Now you are ready to build a Terracotta Lucene application.

Sharing an index

Lets make a quick comparison about the index formats in Lucene (TerracottaDirectory, RAMDirectory and FSDDirectory) to enable you to choose how to cluster your Lucene based application:

	FSDirectory	RAMDirectory	TerracottaDirectory
Index read from	Hard drive	RAM	RAM(but must be faulted from TC server)
Read Speed	Fast	Very Fast	Depends on Network Speed
Memory Footprint	Small	Depends on index size	Small
Pre-instrumentation	No	No	Yes (with Compass TIM)

We recommend using TerracottaDirectory. RAMDirectory integration is not supported.

Now we will share a TerracottaDirectory index which is similar to RAMDirectory. In my opinion, this class is better since a TerracottaDirectory object is broken into many small buckets. We can thus take advantage of partial faulting by Terracotta. i.e. instead of sending a whole index, the Terracotta Server sends only the required buckets to its clients. Moreover, thanks to pre-instrumentation, we can cluster this class without any extra configuration. (Further information can be found in the Compass documentation)

In my code, to avoid extra configuration for our application, I'm using ConcurrentHashMap, which is also pre-instrumented by Terracotta.

Sample Code

Let's look at a small sample application:
The first class, called SharedTerracottaDirectories, has a shared map which is declared as a root in tc-config.xml, containing some indexes (that means these indexes are shared too).
In addition, I have two static methods:

- addDoc(): add a document to an index.
- search(): search a word from an index.

```
public class SharedTerracottaDirectories {
        // This is the shared map where index will be stored.
        public static ConcurrentHashMap<String, Directory> indexMap = new ConcurrentHashMap<String,
Directory>();

        public static void addDoc(Directory directory, String text, Boolean create) {
                try {
                        // Adding a document to the local index
                        IndexWriter writer = new IndexWriter(directory,
                                        new StandardAnalyzer(), create);
                        Document doc = new Document();
                        doc.add(new Field("fieldname", text, Field.Store.YES,
                                        Field.Index.TOKENIZED));
                        System.out.println("Adding this doc --" + doc.get("fieldname")
                                        + "-- to the index");
                        writer.addDocument(doc);
                        writer.optimize();
                        writer.close();
                } ...
        }

        public static void search(Directory directory, String text) {
                // Now search the index:
                try {
                        IndexSearcher searcher = new IndexSearcher(directory);

                        // Parse a simple query that searches for the string "text":
                        QueryParser parser = new QueryParser("fieldname",
                                        new StandardAnalyzer());
                        Query query = parser.parse(text);
                        Hits hits = searcher.search(query);
                        // Iterate through the results:
                        System.out.println("Search results for \""+text+"\":");
                        for (int i = 0; i < hits.length(); i++) {
                                Document hitDoc = hits.doc(i);
                                System.out.println(hitDoc.get("fieldname"));
                        }
                        // Close the index
                        searcher.close();
                        directory.close();
                } ...
        }
}
```

We must declare the map as root to make it shared:

```
<root>
 <field-name>demo.SharedTerracottaDirectories.indexMap</field-name>
</root>
```

Now run Main1:

- We create a TerracottaDirectory index.
- Then we add a document to this index.
- Finally, share the index across the cluster by putting it into a shared map.

```
public class Main1 {
        public static void main(String[] args) {
                TerracottaDirectory directory = new TerracottaDirectory();
                SharedTerracottaDirectories.addDoc(directory, "Hello Terracotta", true);
                SharedTerracottaDirectories.indexMap.put("1", directory);
        }
}
```

Now let's run Main2:

- We retrieve the index from the shared map.
- Add a new document.
- Then search from the new index.

The search result is like we expected. It means that the shared index is working well because we can add new documents to it and search from it between JVMs.

```
public class Main2 {
        public static void main(String[] args) {
                Directory directory = SharedTerracottaDirectories.indexMap.get("1");
                SharedTerracottaDirectories.addDoc(directory, "Bonjour Terracotta",
                                false);
                SharedTerracottaDirectories.search(directory, "Terracotta");
        }
}
```

You can see it's really simple because a lot of things are pre-configured in the Compass TIM.

Master/Worker model - Sharing queries/results

Building a Master/Worker model may speed up a Lucene application. In that case, we need to share queries and results. I won't go in the details but only show you that it is possible.
Again I write a class named SharedQueryAndHitCollector which contains two shared maps:

- One for queries.
- One for HitCollector (sort of Lucene query result).

```
public class SharedQueryAndHitCollector {
        public static ConcurrentHashMap<String, Query> queryMap = new ConcurrentHashMap<String, Query>();
        public static ConcurrentHashMap<String, HitCollector> hitCollectorMap = new ConcurrentHashMap<String,
HitCollector>();
}
```

So I add these roots to the tc-config:

```
        <root>
          <field-name>demo.SharedQueryAndHitCollector.queryMap</field-name>
        </root>
        <root>
          <field-name>demo.SharedQueryAndHitCollector.hitCollectorMap</field-name>
        </root>
```

You can see that we manage the queries and the results like we deal with POJOs.

I also make a class named MyHitCollector which is inherited from HitCollector. This class simply counts the number of hit documents.

```
public class MyHitCollector extends HitCollector {
        AtomicInteger hit = new AtomicInteger();
        @Override
        public void collect(int arg0, float arg1) {
                // TODO Auto-generated method stub
                hit.incrementAndGet();
        }
...
}
```


AtomicInteger is also pre-instrumented so no extra configuration is required.

In the Main4 programs, I've shared two queries:

- A term query.
- A MatchAllDocsQuery query.

```
public class Main4 {
        public static void main(String[] args) {
                // Making 2 queries
                QueryParser parser = new QueryParser("fieldname",
                                new StandardAnalyzer());
                Query q1 = null;
                try {
                        q1 = parser.parse("Hello"); // This query looks for documents having
                        // the word "Hello"
                } ...

                Query q2 = new MatchAllDocsQuery(); // This query will match all
                                                                                        // documents

                // Sharing 2 queries
                SharedQueryAndHitCollector.queryMap.put("1", q1);
                SharedQueryAndHitCollector.queryMap.put("2", q2);

                // Making 2 MyHitCollectors for 2 queries
                MyHitCollector h1 = new MyHitCollector();
                MyHitCollector h2 = new MyHitCollector();

                Directory directory = SharedTerracottaDirectories.indexMap.get("1");
                try {
                        IndexSearcher searcher = new IndexSearcher(directory);
                        // Execute the first query and stock the result into the first
                        // MyHitCollector
                        Query q = SharedQueryAndHitCollector.queryMap.get("1");
                        searcher.search(q, h1);
                        // Execute the second query and stock the result into the second
                        // MyHitCollector
                        q = SharedQueryAndHitCollector.queryMap.get("2");
                        searcher.search(q, h2);

                        // Sharing 2 HitCollectors
                        SharedQueryAndHitCollector.hitCollectorMap.put("1", h1);
                        SharedQueryAndHitCollector.hitCollectorMap.put("2", h2);
                } ...
                System.out.println(h1 + " for query 1");
                System.out.println(h2 + " for query 2");
        }
}
```

Then I execute these two queries and stock the result in two MyHitCollector objects. Finally, I can share these two MyHitCollectors in my cluster. You can check all these shared objects in the Terracotta Developer Console:

- 1 hit document for the first query.

- 2 hit documents for the second query.

About the configuration, I only have to instrument the shared classes:

```
<include>
  <class-expression>org.apache.lucene.search.Query</class-expression>
</include>
<include>
  <class-expression>org.apache.lucene.search.TermQuery</class-expression>
</include>
<include>
  <class-expression>org.apache.lucene.index.Term</class-expression>
</include>
<include>
  <class-expression>org.apache.lucene.search.MatchAllDocsQuery</class-expression>
</include>
<include>
  <class-expression>demo.MyHitCollector</class-expression>
</include>
<include>
  <class-expression>org.apache.lucene.search.HitCollector</class-expression>
</include>
```

Now we've explored how to cluster Lucene via Terracotta.

If you have any questions, suggestions ... I can be reached at neochu1983@yahoo.com

Quartz

Quartz is Now a Terracotta Product

Quartz is now a Terracotta product. The following information pertains to older versions of Quartz and is not recommended unless you are bound to a pre-1.7 version of Quartz. For the latest information on integrating Terracotta and Quartz, see the

Error rendering macro 'html'

Notify your Confluence administrator that "Bob Swift Atlassian Add-ons - HTML" requires a valid license. Reason: EXPIRED

.

Versions Supported	1.5.1, 1.6.1 _RC

Overview

Quartz is a open-source job-scheduling system that can be integrated with or used along side virtually any J2EE or J2SE application. Quartz can be used to create simple or complex schedules for executing tens, hundreds, or even tens-of-thousands of jobs; jobs whose tasks are defined as standard Java components or EJBs.

Quartz terracotta integration enables Quartz clustering based on RAMJobStore. Compared to existing JobStoreTx or JobStoreCMT DB based clustering, Terracotta based RAMJobStore clustering is much more efficient and easy to configure.

Configuration Module

The quickest and easiest way to cluster Quartz is to include Quartz Configuration Module in your Terracotta XML configuration. Add the following Quartz-specific jar file to `tc-config.xml`:

```
<clients>
    <modules>
         <repository>./modules</repository>
         <module name="tim-quartz-1.5.1" />
    </modules>
  </clients>
```

To integrate Quartz 1.6.1, use `tim-quartz-1.6.1`.

Then run the following command from ${TERRACOTTA_HOME} to automatically install the correct version of the TIM:

UNIX/Linux

```
[PROMPT] bin/tim-get.sh install-for <path/to/tc-config.xml>
```

Microsoft Windows

```
[PROMPT] bin\tim-get.bat install-for <path\to\tc-config.xml>
```

What is supported

1. A clustered version of Quartz RAMJobStore is available when using tim-quartz along with Terracotta.

2. Job executions are load balanced across cluster nodes: Each Scheduler instance in the cluster attempts to fire scheduled triggers as fast as the Scheduler permits. The Scheduler instances compete for the right to execute a job by firing its trigger. When a trigger for a job has been fired, no other Scheduler will attempt to fire that particular trigger until the next scheduled firing time.

3. Recovery from failed scheduler instances: When a Scheduler instance fails, Terracotta detects it immediately and Jobs are recovered to be executed by another scheduler at their next execution time.

4. Jobs and trigger information survive node failures: If Terracotta server is configured to run in persistent mode, information is maintained across server restart. i.e. A job triggered on one node, can resume its execution on some other node, if the current node comes down or dies.

5. Recoverable jobs are executed immediately. If their scheduler fails, they are executed by another scheduler instance in the cluster.

Comparison with DB based clustering from Quartz

1. The Terracotta implementation is much faster as everything works in RAM (as compared to DB approach used in JobStoreTx and JobStoreCMT). Terracotta field level changes makes changes to Job and Triggers lightening fast.

2. Hassle free. No DB setup is required.

3. Failed jobs are recovered immediately as compared to DB based approach where scheduler instance are checked for failure based on an interval.

4. No Jgroups or any other cluster configuration is required.

Public Source Repository

SVN: http://svn.terracotta.org/svn/forge/projects/tim-quartz

Private Source Repository (for committers)

SVN: https://svn.terracotta.org/svn/forge/projects/tim-quartz

Rife

Versions Supported	1.6.0

Introduction

RIFE is a full-stack web application framework with tools and APIs to implement most common web features.

Dependencies

None.

Implementation

To integrate Rife 1.6.0, add the following snippet to `tc-config.xml`:

```
<modules>
    <module name="tim-rife-1.6.0" />
</modules>
```

Then run the following command from ${TERRACOTTA_HOME} to automatically install the correct version of the TIM:

UNIX/Linux

```
[PROMPT] bin/tim-get.sh install-for <path/to/tc-config.xml>
```

Microsoft Windows

```
[PROMPT] bin\tim-get.bat install-for <path\to\tc-config.xml>
```

Known Issues

None.

Terracotta Tree Map Cache

Introduction

Terracotta Tree Map Cache, a Terracotta integration module (TIM), is a drop-in replacement for JBoss TreeCache and PojoCache. Terracotta Cache is a distributed cache that is API-compatible with the JBoss TreeCache and PojoCache components. Terracotta Cache can replace these JBoss components with minimal code changes.

Typically, simply changing 1 line of code to instantiate TerracottaCache() instead of TreeCache() is enough because Terracotta Cache maintains most of the JBoss TreeCache APIs.

Dependencies

The following integration modules must be installed with the Terracotta Cache module:

- tim-synchronizedcollection
- tim-synchronizedset
- tim-synchronizedsortedset
- tim-synchronizedmap
- tim-synchronizedsortedmap

See tim-collections for more information.

Implementation

After you download and unpack the synchronized Java collection TIMs listed above into the Terracotta modules directory, add the following to your `tc-config.xml`:

```
<clients>
   <modules>
      <module name="tim-tree-map-cache" />
   </modules>
</clients>
```

Then run the following command from ${TERRACOTTA_HOME} to automatically install the correct version of the TIM:

UNIX/Linux

```
[PROMPT] bin/tim-get.sh install-for <path/to/tc-config.xml>
```

Microsoft Windows

```
[PROMPT] bin\tim-get.bat install-for <path\to\tc-config.xml>
```

Known Issues

- Client applications should declare a `CacheManager` field and mark it as root.
- TerraCotta Cache currently replaces JBoss Cache v.1.4. Due to incompatible API changes, it can not replace JBoss Cache 2.0.

Wicket

Versions Supported	1.3

Apache Wicket is a feature-rich web-application development framework based on Java and HTML. The Terracotta integration module provides for clustering Wicket, primarily user sessions for fail-over.

Richard Wilkinson has created an example clustering a simple Wicket application with Terracotta, available on his blog.

Dependencies

Java 1.5.

Implementation

To integrate Wicket 1.3, add the following snippet to `tc-config.xml`:

```
<modules>
    <module name="tim-wicket-1.3" />
</modules>
```

Then run the following command from ${TERRACOTTA_HOME} to automatically install the correct version of the TIM:

UNIX/Linux

```
[PROMPT] bin/tim-get.sh install-for <path/to/tc-config.xml>
```

Microsoft Windows

```
[PROMPT] bin\tim-get.bat install-for <path\to\tc-config.xml>
```

Known Issues

None.

Grails

Courtesy of ALTERThought. Direct inquiries to ALTERThought.

Overview

This section will show how to cluster Grails using Terracotta. We will cover:

- Generation of a terracotta configuration file
- Generation of terracotta enabled start up scripts for containers

Usage

1. Install the Terracotta plugin

```
grails install-plugin terracotta
```

2. Run the start up script generation task

3. Run the terracotta configuration generation task

4. Generate a war for the application and deploy in a container

5. Start the terracotta server(s)

6. Start the application servers

Generate <ContainerName> Script

Currently only two containers are supported: jboss and tomcat.
Edit TerracottaConfig.properties to set the parameters for the start up scripts: The terracotta install directory on the target jboss or tomcat server

```
terracotta.install.dir
```

The path where the tc.config.xml will be made available on the servers (This plugin does not remote copy it. You have to do it manually)

```
terracotta.config.path
```

Run

```
grails generate-jboss-scripts
```

Copy the generated scripts in the startup script folders of the containers. You will later start the containers using these scripts instead of standard ones.

Generate TcConfig

Generates the tc-config.xml required to run a terracotta enabled container. It enables the clustering of all the domain classes defined in the project and allows to add any additional classes to be made clusterable by including a user defined xml segment. It also clusters the http sessions across the container instances for your application.
Edit TerracottaConfig.properties to set the host names and ports for the terracotta servers to be used by the application.

Edit CustomIncludes.xml to specify any additional terracotta include rules required by your application.

Run

```
grails generate-tc-config
```

Copy the generated file on each container server, in the location specified when generating the container start up scripts.

Running the Clustered Application

Deploy your application on all the container instances. Start the terracotta server(s) Start the containers using the generated start up scripts.

You need a form of load balancing to witness the effects of the clustering, such as Apache with mod_jk.

Test Session Continuity

Start one node of your cluster. Start a session (log in your app, ...). Stop that server instance, then start a second instance. Continue using your application without losing your session!

Additional Documentation

1. The ALTERthought blog
2. The grails blog

Shibboleth

Overview

Shibboleth allows users to securely send trusted information about themselves to remote resources. This information may then be used for authentication, authorization, content personalization, and enabling single sign-on across a broad range of services from many different providers.

IdP Clustering via Terracotta

To cluster IdP, the Shibboleth team recommends using Terracotta to replicate the IdP's in-memory state between nodes.

For fail-over, you should run a Terracotta server on each IdP node. One Terracotta server will run in primary mode while the others run in "hot-standby" (backup) mode. If the primary server fails, the other servers elect a new primary and clients can reconnect to it.

The details for IdP clustering using Terracotta are on Shibboleth's site.

Not Supported at this Time

- Glassbox
- IBM Websphere
- WebSphere CE
- Geronimo

Other Integration Resources

- The TIM Manual – What if your technology isn't listed? Build your own Terracotta Integration Module (TIM)! Also learn how to manage TIM versions in accordance with Terracotta policies.
- Enterprise Services – If a TIM is not available for your technology and you can't devote resources to build your own, we can custom-build it.
- Certified platforms page – A one-page list of certified platforms for the current release.
- The Terracotta Forge – One of the most effective resources available for successfully integrating your project with Terracotta is the Terracotta Forge.