Clustering the Spring Framework

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.

- **Introduction**
- How DSO Clustering Works
- Platform Concepts
- Hello Clustered World
- **Setup and Configuration**
- Planning for a Clustered App
- Configuring Terracotta DSO
- Configuration Reference
- Installation
- **APIs**
- Using Annotations
- Cluster Events
- Data Locality Methods
- Distributed Cache
- Clustered Async Data Processing
- **Tool Guides**
- Developer Console
- Operations Center
- tim-get (TIM Management Tool)
- Platform Statistics Recorder
- Eclipse Plugin
- Sessions Configurator
- Clustering Spring Webapp with Sessions Configurator
- Maven
- JMX
- **Testing, Tuning, and Deployment**
- Top 5 Tuning Tips
- Testing a Clustered App
- Tuning a Clustered App
- Deployment Guide
- Operations Guide
- **FAQs and Troubleshooting**
- General FAQ
- DSO Technical FAQ
- Troubleshooting Guide
- Gotchas
- Non-portable Classes
- **Reference**
- Migrating From DSO
- Concept and Architecture Guide
- Examinator Reference Application
- Clustered Data Structures Guide
- Integrating Terracotta DSO
- Clustering Spring Framework
- Integration Modules Manual
- AspectWerkz Pattern Language
- Glossary

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

Clustering Spring Framework with Terracotta

- Introduction
- Spring Beans
- Spring Sessions
- Spring Web Flow
- Spring Security
- Migrating from a Pre-Terracotta 3.1 Integration

Introduction

Terracotta can cluster applications based on the Spring framework as easily as it clusters any Java application. In addition, direct support for specific Spring technologies such as Web Flow and Security is provided through Terracotta Integration Modules (TIMs).

Terracotta 3.1 and later improved the way Terracotta integrates with Spring. If you've integrated pre-3.1 versions of Terracotta with Spring, see the section on migrating from a previous integration.

For an example of how to use Spring effectively in an application clustered with Terracotta, see the Examinator Reference Application.
For a tutorial on session clustering with the Spring sample web application jpetstore and using the Terracotta Sessions Configurator, see Clustering a Web Application.

Spring Beans

Spring Beans are clustered in the same way as any other POJOs. For example, consider the following bean definition:

```
<bean name="myBean" class="myPackage.MyClass">
  <property name="myInstance" ref="myInstance">
  <property name="dataSource" ref="dataSource"/>
</bean>
```

Case 1: Sharing `myBean`

The `myBean` class, `MyClass`, should be configured for instrumentation in the Terracotta configuration file so that every instance of it (and hence every instance of `myBean`) can be shared:

```
<application>
  <dso>
    <instrumented-classes>
      <include>
        <class-expression>myPackage.MyClass</class-expression>
      </include>
    </instrumented-classes>
  </dso>
</application>
```

Case 2: Declaring a Root in `myBean`

Instead of sharing `myBean`, the `myBean` class, `MyClass`, has a reference `myInstance` you want to share. Furthermore, `myInstance` will be at the root of a graph of shared objects. That reference should be declared a Terracotta root:

```
<application>
  <dso>
    <instrumented-classes>
      <include>
        <!-- Include all classes that will be in the shared object graph
             below myInstance:
          <class-expression>SharedClass</class-expression>
          <class-expression>AnotherSharedClass</class-expression>
          -->
      </include>
    </instrumented-classes>
    <roots>
      <root>
        <field-name>myPackage.MyClass.myInstance</field-name>
      </root>
    </roots>
  </dso>
</application>
```

Case 3: Preventing a Field from Being Shared

If the `MyClass` field `dataSource` should not be shared — for example, if it is specific to each JVM — it should be excluded from instrumentation by being declared transient:

```
<application>
  <dso>
    <transient-fields>
      <field-name>myPackage.MyClass.dataSource</field-name>
    </transient-fields>
  </dso>
</application>
```

Instrumenting a Spring Lookup-Method Bean

You must first add the cglib TIM, then instrument a NoOp proxy class from cglib.

A sample app would probably look something like

```
<bean id="beanB" class="BeanBClass" scope="prototype">
...
</bean>

<bean id="myBean" class="MyClass">
<lookup-method name="createB" bean="beanB"/>
</bean>
```

And then, for MyClass you'd have something similar to:

```
public abstract class MyClass {
protected abstract BeanBClass createB();
}
```

BeanBClass should implement an interface which would be the return type of the `createB()` call. Spring subclasses MyClass into a concrete implementation along with a concrete method implementation for `createB()`.

Spring Sessions

Sessions are clustered by adding the application context name (usually the name of the WAR file) to the Terracotta configuration file:

```
<!-- tc:config/application/dso -->
 <web-applications>
  <!-- list of applications -->
  <web-application>SampleWebApp</web-application>
  ...
 </web-applications>
```

You must configure Terracotta for the specific container you use. See the section on clustering web applications in the Terracotta Product Documentation for more information.

Spring Web Flow

If your application is based on Spring Web Flow, you can cluster it with Terracotta by adding the appropriate TIM.

First, add the following configuration to your Terracotta configuration file:

```
<tc:tc-config xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd">
...
  <clients>
...
    <modules>
      <module name="tim-spring-webflow-2.0" />
    </modules>
...
  </clients>
...
</tc:tc-config>
```

Then install the latest Spring Web Flow TIM for your version of Terracotta by running the following command:

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/tim-get.sh upgrade <path/to/tc-config.xml>
```

Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\tim-get.bat upgrade <path/to/tc-config.xml>
```

The commands above assume that you are using the default name for the Terracotta configuration file, `tc-config.xml`.

Spring Security

If your application is based on Spring Security, you can cluster it with Terracotta by adding the appropriate TIM.

First, add the following configuration to your Terracotta configuration file:

```
<tc:tc-config xsi:schemaLocation="http://www.terracotta.org/schema/terracotta-5.xsd">
...
  <clients>
...
    <modules>
      <module name="tim-spring-security-2.0" />
    </modules>
...
  </clients>
...
</tc:tc-config>
```

Then install the latest Spring Security TIM for your version of Terracotta by running the following command:

UNIX/Linux

```
[PROMPT] ${TERRACOTTA_HOME}/bin/tim-get.sh upgrade <path/to/tc-config.xml>
```

Microsoft Windows

```
[PROMPT] ${TERRACOTTA_HOME}\bin\tim-get.bat upgrade <path/to/tc-config.xml>
```

The commands above assume that you are using the default name for the Terracotta configuration file, `tc-config.xml`.

Migrating from a Pre-Terracotta 3.1 Integration

Beginning with Terracotta 3.1, the <spring> section is no longer supported in the Terracotta configuration file.

If you have integrated your Spring-based application with an earlier version of Terracotta (pre-3.1), the following sections show you how to migrate from that integration.

Migrating from Specialized Spring Beans Configuration

The following example shows how a Spring-based application that had been distributed with Terracotta using Spring-specific Terracotta configuration elements can be distributed using a general configuration approach. The application is still distributed by clustering Spring beans, but now the beans are clustered in the same way as POJOs.

A Spring application context file, `App_context.xml`, contains the following bean definition for the application:

```
<bean id="myBean" class="myPackage.MyClass" />
```

The following application code shows the types that will need to be distributed:

```
public class MyClass {

   @autowired                       // This class is to be injected by Spring.
   private MyInstance myInstance;  // myInstance contains application state that
                                  // must be clustered.

}

/*****/

public class MyInstance {

   // X, Y, and Z represent the state that should be clustered.

   private X x;
   private Y y;
   private Z z;


}

/*****/

public class Main {

// The application is run, with the bean being instantiated.

  public static void main(String[] args) {

    ApplicationContext context = new ClasspathXmlApplicationContext ("/App_context.xml");

    MyClass myBean = context.getBean("myBean") ;  // MyClass instantiated and injected with myInstance.

  }

}
```

In pre-3.1 versions of Terracotta, the application state represented by x, y, and z in MyInstance is clustered by configuring `myBean` in `tc-config.xml`:

```
...
<application>
...
  <spring>
    <jee-application>
    ...
      <application-contexts>
        <application-context>
          <paths>
            <path>*/App_context.xml</path>
          </paths>
          <beans>
            <bean name="myBean">
            ...
            </bean>
          </beans>
       </application-context>
     </application-contexts>
    ...
    </jee-application>
  </spring>
...
</application>
```

Beginning with Terracotta 3.1, the configuration elements shown above are **no longer supported**. To cluster the fields in MyInstance with Terracotta 3.1 and later:

- Use the Terracotta configuration to declare `MyClass.myInstance` to be a Terracotta root, and
- instrument the classes that will be shared in the object graph under `MyClass.myInstance`.

The Terracotta configuration file should then contain the following:

```
<!-- tc:config/application/dso -->
<instrumented-classes>
  <include>
    <class-expression>myPackage.X</class-expression>
    <class-expression>myPackage.Y</class-expression>
    <class-expression>myPackage.Z</class-expression>
  </include>
</instrumented-classes>
<roots>
  <root>
    <field-name>myPackage.MyClass.myInstance</field-name>
<!-- The root name is optional but can help debug. -->
    <root-name>myBean</root-name>
  </root>
</roots>
```

Alternatively, the instrumented classes can also be declared using annotations.

- Add @Root to the field declaration of `myInstance`.
- Add @InstrumentedClass to the classes X, Y, and Z.

> **More Information**
>
> See Terracotta Annotations for more information on using Terracotta annotations.

As a Terracotta root, `myInstance` will behave differently in the cluster than it did when it was clustered by the specialized Terracotta Spring configuration. For example, `myInstance` will become "superstatic," allowing it to persist beyond the lifecycle of any single JVM. If this superstatic behavior is not desired, then a wrapper (one level up) must be created to hold `MyInstance`. See the Terracotta Concept and Architecture Guide for more information on Terracotta roots.

Any class instrumentation and locking done in the <spring> section of the original Terracotta configuration file should be moved to the generic <instrumented-classes> and <locks> sections of the new Terracotta configuration file. When you run Terracotta with the new configuration, note any TCNonPortableObject and UnlockedSharedObject exceptions thrown — these indicate classes that require instrumentation and missing locks. Your main aim should be to discover the application state that requires sharing. See Configuring DSO for more information on how to resolve these exceptions.

Migrating from Specialized Spring Web-Application Configuration

In pre-3.1 versions of Terracotta, you use the name of the WAR file you want to share:

```
<!-- tc:config/application/spring -->
<!-- list of applications -->
<jee-application name="SampleWebApp">
  ...
</jee-application>
```

With Terracotta 3.1 and later, this configuration should look like the following:

```
<!-- tc:config/application/dso -->
 <web-applications>
  <!-- list of applications -->
  <web-application>SampleWebApp</web-application>
  ...
 </web-applications>
```

There is no need to explicitly enable sessions; however, you must configure Terracotta for the specific container you use. See the section on clustering web applications in the Terracotta Product Documentation for more information.

No Support for Spring Events

If you clustered Spring Events in pre-3.1 versions of Terracotta, you had configuration similar to the following in the Terracotta configuration file:

```
<!-- tc:config/application/spring/jee-application/application-contexts/application-context -->
<distributed-events>
  <distributed-event>org.comp.SomeEvent</distributed-event>
  <distributed-event>*.SomeEvent</distributed-event>
  <distributed-event>org.comp.Some*</distributed-event>
  <distributed-event>org.comp.*</distributed-event>
  <distributed-event>*</distributed-event>
</distributed-events>
```

Clustering Spring Events is not supported in Terracotta 3.1 and later. If you have a requirement to cluster Spring Events, bring your questions to the Terracotta Forums for help.