JBossTS 4.4.0

Server Integration Guide

JBossTS-IG-8/18/08

Legal Notices

The information contained in this documentation is subject to change without notice.

Java™ and J2EE is a U.S. trademark of Sun Microsystems, Inc. Microsoft® and Windows NT® are registered trademarks of Microsoft Corporation. Unix is used here as a generic term covering all versions of the UNIX® operating system. UNIX is a registered trademark in the United States and other countries, licensed exclusively through X/Open Company Limited.

Arjuna is a trademark of Hewlett-Packard Company.

Copyright

Software Version

JBossTS 4.4.0

Restricted Rights Legend

Contents

Table of Contents

About This Guide.....4

What This Guide Contains.....4
Audience.....4
Documentation Conventions.....4
Contacting Us.....5

Introduction.....6

Components.....6
Further Reading.....7

Integration Considerations.....8

Configuration mechanism.....8
Logging System.....9
Background processes.....9
TCP/IP Sockets.....10
Using XAResources.....10
Crash Recovery.....11
CORBA ORB.....11
XTS.....12

Server specific Information.....13

JBossTS JTA on JBossAS.....13
JBossTS JTS on JBossAS 14
XTS on JBossAS.....14
JBossTS JTA in Apache Tomcat.....15
JBossTS JTS in Apache Tomcat.....15
JBossTS JTA in Spring Framework.....16
JBossTS JTS in Spring Framework.....17

About This Guide

What This Guide Contains

The Server Integration Guide describes usage of JBossTS for embedding within an application server or framework environment, with particular focus on JBossAS.

Audience

This guide is most relevant to developers and administrators who are responsible for building and managing transactional applications that utilize JBossTS 4.4.0 within an application server.

Documentation Conventions

The following conventions are used in this guide:

Convention	Description
Italic	In paragraph text, italic identifies the titles of documents that are being referenced. When used in conjunction with the Code text described below, italics identify a variable that should be replaced by the user with an actual value.
Bold	Emphasizes items of particular importance.
`Code`	Text that represents programming code.
Function | Function	A path to a function or dialog box within an interface. For example, “Select File | Open.” indicates that you should select the Open function from the File menu.
( ) and |	Parentheses enclose optional items in command syntax. The vertical bar separates syntax items in a list of choices. For example, any of the following three items can be entered in this syntax: `persistPolicy (Never | OnTimer | OnUpdate | NoMoreOftenThan)`
Note: **Caution**:	A note highlights important supplemental information. A caution highlights procedures or information that is necessary to avoid damage to equipment, damage to software, loss of data, or

	invalid test results.

Table 1 Formatting Conventions

Contacting Us

Questions or comments about JBossTS 4.4.0should be directed to our support team.

Chapter 1

Introduction

The JBoss Transaction Service, JBossTS, provides transaction management capability for Java applications. It can be used standalone, or embedded within an application server or other framework/container. This guide is focused on embedded usage.

JBossTS, formerly Arjuna Transaction Service, was acquired by JBoss in 2005 and made open source (LGPL) shortly thereafter. It has been the default transaction manager for JBoss Application Server, JBossAS, since JBossAS 4.2.

Components

JBossTS consists of a transaction engine, ArjunaCore, with multiple 'personalities' layered around it.

JBossTS JTA provides a JTA 1.1 compliant transaction manager for Java Enterprise Edition applications. Transaction scope is limited to a single JVM. This is suitable for use in applications that do not require transaction context propagation on business method calls between JVMs. For example, deployments in which only a single application server instance is used or where several such instances are used for load balancing only, with no transactional communication between them.

JBossTS JTS provides CORBA based distributed transaction management that may be driven either though JTS native API or though the JTA interfaces. It is suitable for situations in which transaction context must span multiple JVMs, such as a JavaEE application deployed on a application server cluster, or where interoperability is required for transactional business method calls between heterogeneous Java application servers or Java and legacy applications written in another language with CORBA bindings.

JBossTS XTS provides transaction support for Web Service applications. Implementing the WS-Atomic Transaction and WS-BusinessActivity protocols, it allows for transactions to span multiple Web Services.

Note that use of transactional resource managers running in processes outside the JVM, e.g. databases or message queues on separate servers, does not necessarily require JBossTS JTS. For transactions spanning multiple resource managers, but where the transactional business logic in Java exists within an single JVM, the JBossTS JTA version is sufficient. Only where transactional business method calls are made between JVMs, is JBossTS JTS needed.

In JBossTS documentation and discussion forums, we occasionally use the terms JTA and JTS to refer to the module of JBossTS i.e. the implementation, rather than to the specification/API of the same name. It it usually clear from the context which usage is intended, but this can occasionally confuse the

unwary. In this document reference to the implementation is consistently prefixed with the product name to avoid confusion i.e. 'JBossTS JTA' is an implementation, 'JTA' is an API specification

Further Reading

More on the history of the product can be found at http://www.arjuna.com/node/29

More on the modular architecture of JBossTS can be found in a series of blog posts at

http://jbossts.blogspot.com/2007/07/trying-to-put-jbossts-into-perspective.html
http://jbossts.blogspot.com/2007/07/trying-to-put-jbossts-into-perspective_13.html
http://jbossts.blogspot.com/2007/07/trying-to-put-jbossts-into-perspective_2937.html
http://jbossts.blogspot.com/2007/07/trying-to-put-jbossts-into-perspective_6770.html

Each of the components of JBossTS has its own extensive documentation set. It is strongly recommended that users take the time to read the manuals for the components they select. This guide does not provide the same level of depth and is not a replacement for the full product documentation. All users should additionally read the ArjunaCore documentation. Although applications may not make use of the Core API (e.g. Transactional Objects) directly, elements of the functionality are used implicitly by the other components of JBossTS. This is particularly true of crash recovery support.

When using JBossTS inside a Java EE application server, the JTA/JTS capability is most likely to be accessed in a specification compliant manner from e.g. web applications, EJBs or other JEE components. Users are expected to be familiar with the relevant specifications, usage patterns and such, and these will not be covered here. Novice Java EE programmers are recommended to peruse suitable additional reading, such as Sun's Java EE tutorial http://java.sun.com/javaee/5/docs/tutorial/doc/ Those new to transaction concepts may find the additional material in the JBossTS Programmer's Guides helpful. The book 'Java Transaction Processing: Design and Implementation' by Mark Little, Jon Maron and Greg Pavlik is also recommended.

The following sections of this guide deal with product specific behavior and configuration points likely to be of interest to developers and administrators. The first chapter covers general concepts relevant to all users. Further sections may be relevant only to integration with particular products and can be skipped by other users.

Integration Considerations

Embedding JBossTS into an existing application framework, IOC container, application server or other similar construct requires the user to take certain points into account. We provide full integration for JBossAS and have less feature rich solutions for Apache Tomcat and Spring. These points are common to any such environment using JBossTS. In the case of JBossAS users they are for information only, the necessary integration steps having already been performed for you.

Configuration mechanism

JBossTS components are configured by properties in one or more xml files. With the exception of XTS, these files are searched for in: user.dir, user.home, java.home and in directories contained on the classpath. System properties may be used to override individual property values or to specify property file locations.

Common errors include inadvertently having multiple JBossTS property files on the search path, or having no such file. In the former case the first suitable file found using the search order provided above will be used. In the latter case, hardcoded defaults will be used for all properties. Neither of these is likely to be desirable and care should be taken that the properties file is correctly located. Where a server is started with a shell script or similar, this should be used to place the directory containing the properties file (not the file itself) on the classpath. The setup-env.sh scripts provided with JBossTS JTA and JTS show examples of this. Where .jar, .war, .ear or other archive files are used for packaging the application, care should be taken these do not contain, or reference though their meta-data, multiple copies of the configuration file.

The most critical configuration property is the location of the ObjectStore. This is a directory hierarchy used to contain the transaction logs. Logging of transaction data is vital to crash recovery, but forms a substantial part of the performance overhead. For production environments the ObjectStore directory should be placed on local, high-performance, fault tolerant RAID devices whenever possible. The default location for this directory is a relative path and unlikely to be desirable. It should changed in the properties file or overridden programmatically.

The classes for JBossTS are contained in a number of .jar files. There are distinct .jar files for the JTA and JTS implementations. One one set should be on the classpath, along with the corresponding properties file. Incorrect behavior may result from mixing JBossTS JTA and JTS .jar files and properties file, or having both on the classpath. In environments that also utilise XTS, care should be taken to run only .jar files from the same release. It is not recommended to use different versions of the JTA or JTS implementation components with the XTS component.

The product's .jar files have dependencies on 3rd party components including other .jar files and configuration files. For binary distributions the product .jar files are in lib and the 3rd party jar files are in lib/ext. These will need adding to the classpath of the server embedding JBossTS. Note that the JBossTS JTS distribution contains a specially patched version of JacORB. It it not recommended to use it with any other version due to issues with crash recovery support.

Logging System

JBossTS uses a logging abstraction layer, the common logging framework, to allow operation with most major logging frameworks. Log4j is preferred. Where this is used, a suitable configuration file should be present. See the common logging framework manual and Log4j documentation for details. Note that for historical reasons the JBossTS code is in the com.arjuna package hierarchy, not org.jboss. This is important when configuring e.g. Log4j thresholds on a per-package basis.

In addition to any filtering done by the chosen 3rd party logging framework, JBossTS uses its own internal logging configuration. This provides very fine grained control over what log statements are printed. However, it is also a source of confusion to many first time users. Log statements must pass filter checks in both JBossTS and the chosen 3rd party logging framework in order to be printed. Setting only one of these to the desired threshold is a common configuration error.

To enable debug level logging from the JBossTS code, edit the properties file such that the com.arjuna.common.util.logging.DebugLevel setting is changed from 0x00000000 to 0xffffffff. Although it is possible to leave this setting in place permanently and rely solely on the log4j threshold to hide debug log output when not required, this is not recommended for performance reasons.

Background processes

JBossTS uses background threads for a number of system tasks. The most obvious of these is processing transaction timeouts. Where a transaction is started with a timeout set, it will be monitored and automatically rolled back by the transaction reaper if it exceeds its specified lifetime. The threads used to manage this behavior are started automatically on demand. However, is some cases it is desirable to start the reaper thread explicitly at server startup time, in order that it inherits a well defined classloader. On-demand startup will cause it to have the classloader of the first application to run a transaction, which is not always desirable with e.g. webapp classloaders.

Is is important to note that a transaction timeout does not interrupt the business logic thread on which the transaction is running. This is done to ensure that long running, non-interruptable business logic operations, such as socket I/O in a database driver, do not block transaction timeouts. This has significant implications for transactional business logic and especially for JCA containers or other connection managers.

In the case of a timeout, the transaction remains bound to the business logic thread but changes state from ACTIVE to ROLLED_BACK. Any use of transactional resource managers that may potentially span this transition must take appropriate precautions for thread safety and to ensure appropriate transaction semantics. For example, a JCA must ensure that any transactional Connections given out to an application thread are invalidated, since the application cannot otherwise detect that the transaction context in which work on the Connection will occur has changed.

Transaction timeouts can be detected in the system log files by entries of the form “Abort of action id <hex string> invoked while multiple threads active within it.”

The application, or container in the case of e.g. EJB CMT, must always call commit or rollback on a transaction it begins, even if that transaction has timed out and been rolled back by the reaper thread. Failure to do so will leave the transaction context bound to the thread. Since nesting of transaction is prohibited by the JTA, this will prevent another transaction from being begun on the thread. Note that the commit and rollback calls may throw an exception to indicate the transaction has already terminated and that appropriate try/catch behavior should therefore always be used.

The other significant use of background threads is for transaction recovery. Although the recovery manager may be run in a separate process, it commonly runs as a background thread in the same JVM as the transaction manager. Regardless of which configuration is used, care should be taken that only one recovery manager is active on each ObjectStore at any given time. Thus the in-process model is unsuitable if more than one JVM is sharing the same ObjectStore. JTS applications typically use an out of process recovery manager, or have one server node designated to run it. This may be a single point of failure for some cluster configurations and it is recommended to seek advise when designing high availability architectures.

The recovery manager must be started either in its own JVM by e.g. the provided shell script, or by programmatically invoking it at server startup. This is a task best handled by the server rather than the application, due to classloading and lifecycle considerations. The recovery manger is not a demon thread, it must explicitly stopped at server shutdown.

TCP/IP Sockets

The recovery process involves determining if a transaction that is present in the ObjectStore is really in need of recovery, or is merely completing at a slower than expected pace. To achieve this the recovery manager must communicate with the transaction manager. This is done using a simple TCP/IP socket based protocol, even when the components are co-located in the same JVM. Care should be taken to ensure appropriate firewall configuration where necessary.

JBossTS also uses a TCP/IP socket to work around a limitation in the JVM with regard to process identifiers. Globally unique identifiers (Uids) are used extensively by JBossTS. One element used in each Uid is a process identifier, which must be distinct on a per-host basis at any given point in time. Unfortunately the Java platform API does not expose the operating system's process id for the JVM to Java code running inside it. Therefore, JBossTS binds a listener to a random port on the host and uses that port number in place of the process id. Since only one process can listen a given interface-port pair at a the same time, this guarantees uniqueness. However, when running JBossTS in multiple JVMs on the same host, care should be taken that the same interface is used by each JVM for this purpose. Whereas server threads listening for client requests may bind to a specific address on systems with more than one interface, it is strongly recommended that this listener always use 'localhost'. It is not necessary that inbound requests can reach the listening port, as it never actually processes any.

Using XAResources

When accessed through the JTA, JBossTS operates on XAResources only, with no knowledge of Connections or other such constructs. This has important implications for JCA or connection management frameworks. In addition to the points discussed previously regarding asynchronous processing of

transaction timeouts, it requires that the JCA properly manage aspects such as the last resource gambit and transaction recovery.

Transaction managers such as JBossTS are designed to operate chiefly in conjunction with XA aware resource managers e.g. databases and message servers that have distributed transaction capability. However, it is occasionally necessary or desirable to include within a transaction, a resource manager that is not XA capable. This may be the case where a database has no XA driver available. In such situations it is possible to provide JBossTS with an XAResource implementation that calls commit on the connection within the resource's prepare method and implements the resource's commit method as a null-op. By telling JBossTS to order this last in the sequence of resources called during transaction termination, something close to XA transaction behavior is possible. This is known as the last resource gambit or last resource commit optimization. By default it is considered an error to enlist more than one such resource in a transaction, as doing prevents the transaction from providing the expected ACID properties. However, for application that desire such behavior, a configuration options exists to enable it. This is a server wide setting and should be used with care, as it may allow applications requiring full ACID semantics to be accidentally deployed with a broken configuration.

Crash Recovery

Recovery of crashed transactions ensures eventual termination of the transaction even in the event of server or network failures. The ObjectStore contains a serialized snapshot of the transaction state, which is used to recreate it on server restart and retry the transaction termination. However, there is a timing window in which a crash will result in a resource manager holding locks for an in-doubt transaction which is not recorded in the ObjectStore. Further, even when the transaction is recorded ,that record may not contain all the required information. This is particularly the case when using drivers that have non-serializable XAResource implementations, which unfortunately is most of them.

To address these situations, JBossTS uses configurable recovery modules. By providing a suitable plugin for any resource manager that is used, it can be ensured that all transactions will be recovered successfully. Without such configuration, the system will keep retrying failed transactions repeatedly without success. This leads to three consequences: resource managers may remain locked, denying service to other applications until such time as an administrator intervenes; entries for the failed transactions may remain in the ObjectStore indefinitely; the recovery manager will log failed recovery attempts on each pass, leading to larger than necessary log files. These situations commonly manifest themselves by repeated log entries of the form “Could not find new XAResource to use for recovering non-serializable XAResource < id string >” Such errors can be ignored in development environments and eliminated by shutting down JBossTS and removing the contents of the ObjectStore before restarting. However, it is important to realize such log entries are indicative of misconfiguration and should be a serious concern in a production environment.

CORBA ORB

JBossTS JTS component provides distributed transactions using a CORBA ORB transport. It includes an orb portability layer, facilitating porting to any specification compliant ORB. Please consult the ORB Portability Guide for further information.

When using JacORB, which is the preferred ORB for the product, a specially patched version is required. This is shipped in the jacorb directory of the distribution. Correct behavior, particularly for crash recovery, is not guaranteed on other versions of JacORB.

Certain JacORB configuration options are critical to the operation of JBossTS. In particular, it may require a larger than normal poa thread pool for reasonable performance. A value of 32 is suggested, rather than the default 8. The jacorb.implname must be unique for each JacORB instance. This requires each to have its own config file, or for the value to the overridden in code.

XTS

The Web Services transaction component of JBossTS implements the WS-AT and WS-BA specifications. It requires code from the JBossTS JTA or JBossTS JTS and one of these must therefore be present in the JVM even if not used directly. The reverse is not true: XTS is optional and need not be deployed for the JTA and JTS components to function.

When deploying XTS, care should be taken to ensure version compatibility with other components of JBossTS that may already be present in the server. In general it is recommended to deploy only the release version of XTS exactly matching the JBossTS JTA or JTS release used.

XTS implements the 1.0 version of the protocols using its own embedded SOAP stack, and the 1.1 version of the protocols using JBossWS. JBossWS provides an abstraction layer over a number of SOAP stacks incuding JBossWS native, glassfish's Metro stack and Apache CXF. Of these, only JBossWS native is fully functional at this time. The 1.0 and 1.1 versions of XTS may be deployed individually or together. However, the two protocols do not interoperate.

Embedding XTS into application servers other than JBossAS is a highly involved task. It is likely to be possible to embed the 1.0 version more easily, as this uses its own lightweight SOAP stack based on a servlet implementation. Thus any servlet container should suffice. The 1.1 version is less portable, currently requiring JBossAS as the container.

It should be noted that the SOAP stacks mentioned above are used only for propagation of transaction control messages. Business logic calls between Web Services may flow over any stack. Transaction context interceptors are provided for JAX-RPC and JAX-WS.

Server specific Information

This chapter consists of sections detailing integration information specific to selected frameworks or servers. Uses need only consult the following sections relevant to them. In all cases however, knowledge of the material in the preceding chapters is assumed.

JBossTS JTA on JBossAS

Starting with JBossAS 4.2, the JBoss application server comes with JBossTS JTA pre-integrated. This documentation applies to JBossAS version 5.0.

It is not normally necessary to upgrade the JBossTS component independently of the application server, but this may be done by replacing the appropriate .jar files (in server/.../lib) and config file (server/.../conf/jbossjta-properties.xml) if required. Customers using JBoss EAP should upgrade only on the advise of JBoss support staff.

The transaction manager is deployed into the application server via server/.../deploy/transaction-beans.xml This file overrides the ObjectStore directory setting present in the jbossjta-properties.xml file.

Logging is configured by a combination of the settings in server/.../conf/jbossjta-properties.xml and server/.../conf/jboss-log4j.xml Refer to the previous chapter for further information on the logging mechanism used by JBossTS.

By default any XA datasource configured via JBossAS JCA, i.e. those deployed though -ds.xml files with <xa-datasource> elements, is automatically registered for crash recovery. Non-XA datasources i.e. those using <local-tx-datasource> are enlisted in transactions using the last resource gambit. Thus by default only one such datasource may be used per transaction.

The Transactional Driver provided by JBossTS is redundant in an application server environment. The JCA should be used instead.

Transaction manager configuration information is available though the JBossAS JMX console, using the TODO bean name . However, transaction statistics gathering is turned off by default for performance reasons to the counters will read zero unless explicitly enabled via the config file.

Using the default configuration, transaction boundaries may not span multiple JVMs. Therefore, you should not make e.g. transactional calls to EJBs in another server instance. The exception to this is the limited transaction boundary demarcation support provided by the server's ClientUserTransaction. For transaction use cases involving multiple JVMs, JBossTS JTS is required.

JBossTS JTS on JBossAS

The JTA component which is the default in JBossAS may be replaced by the JTS component by following the steps in the JBossTS JTS INSTALL document.

Because the JTS is CORBA based, it requires a JBossAS configuration that contains a CORBA ORB. Therefore it is suitable for use in the 'all' but not the 'default' server configuration. The version of JacORB shipped with JBossAS already has the patches needed by JBossTS, it is not necessary to upgrade.

Confusion and misconfiguration is common where properties files and/or configuration files for both the JTA and JTS components exist on the classpath or in the same server instances. When replacing the JTA implementation it is important to fully remove it, not merely add the JTS components to the same server. Similarly, do not attempt to use the JBossTS JTA transaction manager bean with the .jar files from the JTS version unless you also carefully edit the properties file. In most cases conflicting configuration information will make this combination unstable and it is not recommended.

JBossAS binds to localhost by default. This is unlikely to be suitable for a distributed transaction environment. It is recommended to use an alternative IP address when running the server with JBossTS JTS.

For Java EE applications, the transaction manager functionality is accessed through the JTA API. Application level code should not perceive any difference between the JBossTS JTA and JBossTS JTS implementations of this API, making the upgrade transparent to them. The exceptions are a) obviously applications making transactional calls to another JVM will have different semantics and b) users of ClientUserTransaction. The client side transaction support does not work with the JTS and should not be used. Client applications may access the transaction manager via CORBA if they require transaction control. They will need to have the JBossTS JTS installed for this.

By default remote business logic calls between JBossAS servers use the JRMP protocol. With suitable configuration they may also use the interoperable RMI/IIOP protocol. Transaction context propagation over both is automatic. However, the transaction control calls will always flow over RMI/IIOP regardless of the protocol used for the business logic calls.

The points regarding logging, XAResources and JMX monitoring/management from the JBossTS JTA section also apply to the JTS version.

XTS on JBossAS

XTS is deployed as a JBoss Service Archive (.sar) on JBossAS. It reuses the JTA or JTS implementation that is already present in the server, so must be of a compatible version.

Automatic crash recovery functionality for XTS is currently available only for the WS-AT protocol, not for WS-BA. Applications requiring such support must implement their own solutions at present.

JBossTS JTA in Apache Tomcat

The JBoss application server already embeds both Tomcat and JBossTS JTA. It is highly modular and can be configured to remove all other components that may not be required, such as the EJB container. Furthermore, it also has a JCA capable of managing XA Datasources. These characteristics make it preferable to Tomcat for most web applications requiring a transaction manager. Nevertheless, we recognize that a small proportion of users and unable to adopt this solution and therefore offer the following guidance for integration of JBossTS JTA with Tomcat.

Bundling JBossTS with a web application is not recommended. The transaction manager should be integrated into the server instead. It behavior when potentially running multiple instances in the same JVM or being reloaded, both of which can occur when bundled with a webapp, is undefined at best.

The JBossTS .jar files, 3rd party pre-requisites and configuration file must be on Tomcat's classpath. This is best achieved by editing the server's startup script.

A server lifecycle listener provides ideal hook points for startup and shutdown of the background processes used by JBossTS, including the transaction reaper and recovery manager.

The UserTransaction can be exposed to web applications via the <Transaction ...> element of their context.xml file.

The connection pool provided with Tomcat is not a JCA. It does not handle XA drivers, nor provide last resource behavior for non-XA ones. Connections obtained from a normal pool will not participate automatically in the JTA transaction.

Alternatives include XAPool, which is not recommended due to limited functionality; driving the XAResource enlistment and crash recovery directly from application code; or using a custom wrapper around the JBossTS Transactional Driver. The latter is the recommended approach, as it requires least additional code and handles complex issues such as asynchronous rollbacks and crash recovery using well established techniques.

At this time the code for integration of JBossTS with Tomcat is not part of the JBossTS release. However, an unsupported prototype implementation is available separately from the project's source code repository at https://svn.jboss.org/repos/labs/labs/jbosstm/workspace/jhalliday/tomcat-integration/

JBossTS JTS in Apache Tomcat

The techniques for integrating the JTS version of JBossTS are very similar to those needed for the JTA version, described in the section above. The major difference is that the JTS component requires an ORB and some additional code to correctly initialize it at server startup time. This can be achieved by including the version of JacORB shipped with JBossTS into the Tomcat classpath and adding appropriate statup code to the lifecycle listener described above.

The same considerations with regard to XA connection management that apply for the JBossTS JTA version also hold true for JBossTS JTS.

No sample code for JBossTS JTS integration is provided at this time. Users are recommended to use the JTA version of the integration code as a starting point to develop their own solution.

JBossTS JTA in Spring Framework

The Spring framework (http://www.springframework.org) allows pluggable transaction managers. If you need XA support for your Spring application then you need to plug in a transaction manager such as JBossTS. For use within the JBossAS application server version 4.2 or later, you probably don't need to do anything. Just ensure your Spring application is configured to use

```
<bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager" />
```

then the JtaTransactionManager? will automatically find and use JBossTS via the application server's JNDI.

For standalone use, i.e. outside the application server, where you need JTA, the configuration is almost as easy:

```
  <bean id="jbossTransactionManager"
class="com.arjuna.ats.internal.jta.transaction.arjunacore.TransactionManagerI
mple" />

  <bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager">

    <property name="transactionManager">

      <ref bean="jbossTransactionManager"/>

    </property>

  </bean>
```

You also need some additional elements in your classpath. Using ant format, these are:

```
  <path id="jbossts">

    <fileset dir="${jbossts.home}/lib">

      <include name="*.jar"/>

    </fileset>

    <fileset dir="${jbossts.home}/lib/ext">

      <include name="*.jar"/>
```

```
      <include name="*.zip"/>

    </fileset>

    <pathelement location="${jbossts.home}/etc"/>

  </path>
```

JBossTS JTS in Spring Framework

If you are one of the people who need full JTS support outside the application server, there is a bit more work involved. Aside from a slightly different configuration in the .xml file and including the ORB classes on the classpath, you also need to call some setup methods in your code to correctly initialize JBossTS before performing any transactions.

config:

```
<bean id="jbossTransactionManager"
class="com.arjuna.ats.internal.jta.transaction.jts.TransactionManagerImple">

</bean>

<bean id="transactionManager"
class="org.springframework.transaction.jta.JtaTransactionManager">

    <property name="transactionManager">

      <ref bean="jbossTransactionManager"/>

    </property>

</bean>
```

classpath:

```
  <path id="jbossts">

    <fileset dir="${jbossts.home}/lib">

      <include name="*.jar"/>

    </fileset>

    <fileset dir="${jbossts.home}/lib/ext">

      <include name="*.jar"/>

      <include name="*.zip"/>

    </fileset>
```

```
          <pathelement location="${jbossts.home}/etc"/>

          <fileset dir="${jbossts.home}/jacorb/lib">

            <include name="*.jar"/>

          </fileset>

        </path>
```

setup code:

```
        com.arjuna.orbportability.ORB myORB =
com.arjuna.orbportability.ORB.getInstance("ServerSide");

        com.arjuna.orbportability.RootOA myOA =
com.arjuna.orbportability.OA.getRootOA(myORB);

        myORB.initORB(new String[] {}, null);

        myOA.initOA();
```

The above guidelines assumes JBossTS 4.2.3.SP3 and Spring 2.0.5. However, neither this combination nor any other is officially supported outside JBossAS at this time.