

JBoss EJB 3.0

JBoss EJB 3.0 Reference Documentation

RC8 - FD

Table of Contents

Preface	iv
1. Session EJB and MDB Configuration	1
1.1. Pooling	1
1.2. The Stateful Session Bean Cache	2
1.2.1. Non-Clustered	2
1.2.2. Clustered	3
1.2.3. No Passivation	3
2. References and Injection XML and Annotations	4
2.1. Intro	4
2.2. EJB references	4
3. @Resource Annotation	5
4. Entity Configuration Files	6
4.1. Intro	6
4.2. Configuration Files	6
4.2.1. Configure JBoss datasource	6
4.2.2. persistence.xml and .jar files	6
4.3. EAR and WAR files	8
4.4. Referencing persistence units	8
4.4.1. Scoped deployments	8
4.4.2. Referencing from Global JNDI	8
4.5. Securing Entity Beans	8
5. Hibernate Integration	11
5.1. Hibernate Mapping Files	11
5.2. Injection Hibernate Session and SessionFactory	11
5.3. Access to org.hibernate.Session	11
5.4. Access to org.hibernate.Query	12
6. JBoss EJB 3.0 extensions	13
6.1. @Service EJBs	13
6.1.1. @org.jboss.annotation.ejb.Management interface	13
6.1.2. Lifecycle Management and dependencies	13
6.1.2.1. Lifecycle Methods	13
6.1.2.2. Dependencies	14
6.1.3. Example	14
6.2. Message Driven POJOs	16
6.3. Asynchronous invocations	16
7. JBoss EJB 3.0 jboss.xml deployment descriptor	19
7.1. Bean extensions	19
7.1.1. Service	19
7.1.2. Consumer/Producer	22
8. JBoss EJB 3.0 partial deployment descriptors	24
8.1. Overview	24
8.2. Examples	24
8.2.1. Complete deployment descriptor	24
8.2.2. Security	26
8.2.3. Transactions	27
8.2.4. References	27
8.2.5. Callbacks	27
9. JBoss EJB 3.0 MDB JCA Inflow	28

9.1. SwiftMQ JCA XA Resource	28
9.2. Non-Standard Messaging Types	28
10. Transports	30
10.1. Default transport	30
10.2. Securing the transport	30
10.2.1. Generating the keystore and truststore	30
10.2.2. Setting up the SSL transport	31
10.2.3. Configuring your beans to use the SSL transport	32
10.2.4. Setting up the client to use the truststore	32

Preface

This isn't the only reference to learn EJB 3.0 Look in the JBoss EJB 3.0 tutorial for loads of example programs and detailed text describing each example.

In some of the example listings, what is meant to be displayed on one line does not fit inside the available page width. These lines have been broken up. A '\' at the end of a line means that a break has been introduced to fit in the page, with the following lines indented. So:

```
Let's pretend to have an extremely \  
    long line that \  
    does not fit  
This one is short
```

Is really:

```
Let's pretend to have an extremely long line that does not fit  
This one is short
```

Session EJB and MDB Configuration

1.1. Pooling

Both Stateless Session beans and Message Driven Beans have an instance pool. The basic configuration of JBoss uses a thread local pool to avoid Java synchronization (`org.jboss.ejb3.ThreadLocalPool`). These EJB types can be configured to use an alternative pooling mechanism. For example, JBoss has a strict pool size implementation that will only allow a fixed number of concurrent requests to run at one time. If there are more requests running than the pool's strict size, those requests will block until an instance becomes available. This is configured via the `@org.jboss.annotation.ejb.PoolClass` annotation.

```
@Retention(RetentionPolicy.RUNTIME)
@Target({ElementType.TYPE})
public @interface PoolClass
{
    Class value();
    int maxSize() default 30;
    long timeout() default Long.MAX_VALUE;
}
```

The `value()` parameter defines the pool implementation class you want to plug in. `maxSize()` defines the size of the pool while `timeout()` is a time in milliseconds you want to block when waiting for an instance to be ready. This annotation can be applied to a stateless or message driven bean class. Here's an example of using it.

```
@Stateless
@PoolClass (value=org.jboss.ejb3.StrictMaxPool.class, maxSize=5, timeout=10000)
@Remote(StrictlyPooledSession.class)
public class StrictlyPooledSessionBean implements StrictlyPooledSession
{
    ...
}
```

There is no nice way of applying the same configuration through XML. To do it through XML you must define a new aspect domain (an EJB container configuration template) and define an annotation override within that domain and then apply the domain through `jboss.xml`. Here's an example:

First create the aspect domain. Create a file called `mydomain-aop.xml` and put this in the top level directory of your EJB jar. Might seem a little cryptic but what this is doing is declaring an annotation that will be created by the EJB container. Our EJB3 implementation is based on JBoss AOP. See the JBoss AOP documentation for more info on annotation overrides.

```
<aop>
  <domain name="Strictly Pooled Stateless Bean" extends="Stateless Bean" inheritBindings="true">
    <annotation expr="!class(@org.jboss.annotation.ejb.PoolClass)">
      @org.jboss.annotation.ejb.PoolClass (value=org.jboss.ejb3.StrictMaxPool.class, maxSize=5, timeout=10000)
    </annotation>
  </domain>
</aop>
```

```

</domain>
<domain name="Strictly Pooled Message Driven Bean" extends="Message Driven Bean" inheritBindings="true">
  <annotation expr="!class(@org.jboss.annotation.ejb.PoolClass)">
    @org.jboss.annotation.ejb.PoolClass (value=org.jboss.ejb3.StrictMaxPool.class, maxSize=5, timeout=1000)
  </annotation>
</domain>
</aop>

```

There is no nice way of applying the same configuration through XML. To do it through XML you must define a new aspect domain (an EJB container configuration template) and define an annotation override within that domain and then apply the domain through jboss.xml. Here's an example:

The next thing you have to do is apply the aspect domain to your EJB within a jboss.xml file in the META-INF directory.

```

<?xml version="1.0"?>
<jboss
  xmlns="http://java.sun.com/xml/ns/javaee"
  xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
  xsi:schemaLocation="http://java.sun.com/xml/ns/javaee
    http://www.jboss.org/j2ee/schema/jboss_5_0.xsd"
  version="3.0">
  <enterprise-beans>
    <session>
      <ejb-name>MySessionBean</ejb-name>
      <aop-domain-name>Strictly Pooled Stateless Bean</aop-domain-name>
    </session>
  </enterprise-beans>
</jboss>

```

1.2. The Stateful Session Bean Cache

Stateful beans are stored in a cache. This cache is responsible for passivated stateful sessions when the cache becomes too full or a bean is too old. You may want to set things like the max size of this cache, and when beans should become idle. Configuration is different depending on whether you are clustered or not.

1.2.1. Non-Clustered

For non clustered stateful beans, the `@org.jboss.annotation.ejb.cache.simple.CacheConfig` annotation is responsible for configuring the cache.

```

public @interface CacheConfig
{
    int maxSize() default 100000;

    long idleTimeoutSeconds() default 300;
}

```

If you want an XML version, you must do a similar pattern as shown in the pooling section above. You must create an aspect domain through XML and apply that domain through XML.

1.2.2. Clustered

For non clustered stateful beans, the `@org.jboss.annotation.ejb.cache.tree.CacheConfig` annotation is responsible for configuring the cache.

```
public @interface CacheConfig
{
    String name() default "jboss.cache:service=EJB3SFSBClusteredCache";

    int maxSize() default 10000;

    long idleTimeoutSeconds() default 300;
}
```

The `name()` attribute specifies the kernel name of the clustered cache you are using. Usually you won't change this value. If you want an XML version, you must do a similar pattern as shown in the pooling section above. You must create an aspect domain through XML and apply that domain through XML.

1.2.3. No Passivation

Sometimes it is useful to turn off passivation entirely. This can be done by plugging in the caching implementation using the `@org.jboss.annotation.ejb.cache.Cache` (`org.jboss.ejb3.cache.NoPassivationCache.class`) annotation.

References and Injection XML and Annotations

2.1. Intro

This section covers how JBoss implements @EJB and @Resource. Please note that XML always overrides annotations.

2.2. EJB references

Rules for the @EJB annotation

- The @EJB annotation also has a mappedName() attribute. The specification leaves this a vendor specific metadata, but JBoss recognizes mappedName() as the global JNDI name of the EJB you are referencing. If you have specified a mappedName(), then all other attributes are ignored and this global JNDI name is used for binding.
- If you specify @EJB with no attributes defined:

```
@EJB ProcessPayment myEjbref;
```

Then the following rules apply:

- The EJB jar of the referencing bean is contained in is search for another EJB with the same interface. If there are more than one EJB that publishes same business interface, throw an exception, if there is one, use that one.
- Search the EAR for EJBs that publish that interface. If there are duplicates, throw an exception, otherwise return that one.
- Search globally in JBoss for an EJB of that interface. Again, if duplicates, throw an exception
- @EJB.beanName() corresponds to <ejb-link>. If the beanName() is defined, then use the same algorithm as @EJB with no attributes defined except use the beanName() as a key in the search. An exception to this rule is if you use the ejb-link '#' syntax. The '#' syntax allows you to put a relative path to a jar in the EAR where the EJB you are referencing lives. See spec for more details

For XML the same rules apply as annotations exception <mapped-name> is the ejb-jar.xml equivalent to @EJB.mappedName().

@Resource Annotation

The `@Resource` annotation can be used to inject env-entries, `EJBContext`, JMS destinations and connection factories, and datasources. For datasources and JMS destinations and connection factories, JBoss uses the `@Resource(mappedName=)` attribute. This attribute corresponds to the global JNDI name of the referenced resource. For example, here's how you would use the `@Resource` annotation to inject a reference to the default datasource:

```
@Resource(mappedName="java:/DefaultDS") DataSource ds;
```

Here's one for a JMS connection factory:

```
@Resource(mappedName="java:/ConnectionFactory") ConnectionFactory factory;
```

Entity Configuration Files

4.1. Intro

This section talks about various configuration parameters for getting entities to work within JBoss. JBoss EJB 3.0 is built on top of the Hibernate ORM solution, and more specifically, the Hibernate Entity Manager whose documentation comes with your JBoss EJB 3.0 distribution. JBoss EJB 3.0 also requires JBoss JDBC connection pools. You'll learn you to configure both and how they relate to eachother.

4.2. Configuration Files

To use Entity beans within JBoss EJB 3.0 you'll need to do a few things.

- Configure a JBoss datasource using *-ds.xml file. Check out our documentation or you can view example configurations for many different datasources within: `jboss-dist/docs/examples/jca`
- Create a persistence.xml file and jar up your entity classes with the persistence.xml file living in the META-INF/ directory.

4.2.1. Configure JBoss datasource

For datasources, JBoss comes with the Hypersonic SQL database embedded within it and a default datasource available in JNDI under `{java:/DefaultDS}`. Otherwise, you'll need to specify your own datasource. Please refer to the JBoss AS guide on how to create a jBoss connection pool. Also, there are examples in the jboss distribution under `docs/examples/jca`.

4.2.2. persistence.xml and .jar files

For those of you familiar with older versions of the spec, there is no .par file anymore. Entities are placed in a EJB-JAR .jar file or a .jar file all their own. You must also define a persistence.xml file that resides in the META-INF folder of the .jar file. Here's an example of a persistence.xml file.

```
<persistence>
  <persistence-unit name="manager1">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <jar-file>../MyApp.jar</jar-file>
    <class>org.acme.Employee</class>
    <class>org.acme.Person</class>
    <class>org.acme.Address</class>
    <properties>
      <property name="hibernate.dialect" value="org.hibernate.dialect.HSQLDialect"/>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
    </properties>
  </persistence-unit>
</persistence>
```

name

You are required to name your persistence unit. If your persistence classes are within a EJB JAR archive, and it is the only persistence unit defined, then you do not have to reference this name explicitly in your `@PersistenceContext` and `@PersistenceUnit` annotations.

jta-data-source, non-jta-data-source

This is the JNDI name of where the `javax.sql.DataSource` is located. This is ignored when `*not*` used within an application server. When running outside of an application server, you must specify JDBC connections with Hibernate specific properties (see below). If you're running inside JBoss, put the jndi name of the datasource you defined in the earlier section. Remember to put the "java:/" in front of the jndi name you selected for your datasource.

jar-file and class

The class element specifies a fully qualified classname that you will belong to the persistence unit. The jar-file element specifies another jar you want automatically scanned for `@Entity` classes. When using jar-file, you must specify a path relative to the jar file the persistence.xml file is in. By default also, the jar the persistence.xml file is placed in is scanned for `@Entity` classes as well.

properties

The properties element is used to specify vendor specific properties. This is where you will define your JBoss and Hibernate specific configurations.

JBoss EJB 3.0 is built on top of Hibernate 3.0 ORM solution. You may need to provide information to Hibernate so that it knows the database vendor dialect (MySQL, Oracle, etc...), caching, as well as other settings. JBoss EJB 3.0 also has some specific configurable properties as well. Here's a table of properties. We don't list all the Hibernate ones. You can go look in the Hibernate documentation for those.

Table 4.1. Example Config Properties

Property	Description
hibernate.dialect	Usually Hibernate can figure out the database dialect itself, but maybe not. Check the hibernate doco for information on this
hibernate.hbm2ddl.auto=update	Creates the database schema on deploy if it doesn't exist. Alters it if it has changed. Useful for when you want to generate database schema from entity beans
hibernate.cache.provider_class	Defines the caching architecture that Hibernate should use. There is a tutorial to set up a clustered cache.
jboss.entity.manager.jndi.name	JBoss does not publish container managed EntityManagers in JNDI by default. Use this to bind it.
jboss.entity.manager.factory.jndi.name	JBoss does not publish container managed EntityManagerFactorys in JNDI by default. Use this to bind it.
jboss.no.implicit.datasource.dependency	JBoss tries to register deployment dependencies for datasource by guessing the dependency name based on the jndi name of the datasource. Use this switch if the guess is wrong.
jboss.depends.{some arbitrary name}	Specify an MBean dependency for the persistence unit deployment.

4.3. EAR and WAR files

JBoss 4.0.x does not support the Java EE 5 EAR format. So, if you want to deploy a standalone persistence archive, you must list it within application.xml as an ejb module.

For WAR files, JBoss 4.0.x does not yet support deploying a persistence archive with WEB-INF/lib as required by the spec.

4.4. Referencing persistence units

4.4.1. Scoped deployments

If a persistence unit is defined in an EJB-JAR file it is not visible to other deployed jars using the `@PersistenceContext` or `@PersistenceUnit` annotation. This scoping is required by the specification.

4.4.2. Referencing from Global JNDI

Persistence units are not available within global JNDI unless you explicitly configure them to do so. There are two properties you can specify in your persistence.xml file to enable this. `jboss.entity.manager.jndi.name` gives you a transaction scoped entity manager you can interact with. `jboss.entity.manager.factory.jndi.name` binds the entity manager factory into global JNDI.

```
<persistence>
  <persistence-unit name="manager1">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="jboss.entity.manager.jndi.name" value="java:/Manager1"/>
      <property name="jboss.entity.manager.factory.jndi.name" value="java:/Manager1Factory"/>
    </properties>
  </persistence-unit>
</persistence>
```

4.5. Securing Entity Beans

There is no spec defined way of security EJB 3.0 entity beans. JBoss does provide a solution but it is only available within the JBoss Application Server. You can secure access to the CRUD operations of your entity, but not method invocations on the entity. Basically, what you are securing is when the Entitymanager does `persist()`, `remove()`, etc. operations on an entity. To secure an entity you must add additional property information within persistence.xml. here's an example:

```
<?xml version="1.0" encoding="UTF-8"?>
<persistence>
  <persistence-unit name="tempdb">
    <jta-data-source>java:/DefaultDS</jta-data-source>
    <properties>
      <property name="hibernate.hbm2ddl.auto" value="create-drop"/>
      <property name="hibernate.jacc.allowed.org.jboss.ejb3.test.jacc.AllEntity" value="true"/>
      <property name="hibernate.jacc.allowed.org.jboss.ejb3.test.jacc.StarEntity" value="true"/>
      <property name="hibernate.jacc.allowed.org.jboss.ejb3.test.jacc.SomeEntity" value="true"/>
      <property name="hibernate.jacc.enabled" value="true"/>
    </properties>
  </persistence-unit>
</persistence>
```

```

        </properties>
    </persistence-unit>
</persistence>

```

The property name has the role and the entity class within it. It is "hiberante.jacc." + role + FQN of entity class.

The value can be one of insert, update, delete, or read. That is the permission. The "*" character can be used to specify all permissions.

Now the configuration gets a bit more complicated. You have to interact with the EntityManager within a secured jacc enabled EJB (session) or secured jacc enabled WAR. For jacc enabling a WAR, see this URL <http://wiki.jboss.org/wiki/Wiki.jsp?page=JACC>. To jacc enable an EJB, you have to use a JBoss specific aspect domain. An aspect domain is a container configuration template. There is an aspect domain defined for jacc, either "JACC Stateless Bean" or "JACC Stateful Bean" depending on the type of EJB you are securing. For example:

```

@javax.ejb.Stateless
@SecurityDomain ("other")
@AspectDomain("JACC Stateless Bean")
public class StatelessBean implements Stateless
{
    @PersistenceContext
    EntityManager em;

    @PermitAll
    public int unchecked(int i)
    {
        System.out.println("stateless unchecked");
        return i;
    }

    @RolesAllowed ("allowed")
    public int checked(int i)
    {
        System.out.println("stateless checked");
        return i;
    }
}

```

You must do similar configuration on Tomcat to jacc enable wars and your servlets.

The final thing you must do is to create an jacc-service.xml file that will go in your deploy/ejb3.deployer directory. This jacc configuration is defined in more detail at <http://wiki.jboss.org/wiki/Wiki.jsp?page=JACC>. The jacc-service.xml file must be deployed before your EJB3 Deployer. Putting this file in the ejb3.deployer directory ensures that this service starts before the EJB3 subsystem. You could also set up an explicit dependency on the jacc service within the ejb.deployer/META-INF/jboss-service.xml file if you do not like that kind of setup. The EJB 3.0 distribution comes with a tutorial on entity security.

```

<server>
  <mbean code="org.jboss.security.jacc.DelegatingPolicy"
    name="jboss.security:service=JaccPolicyProvider"
    xmbean-dd=" ">
    <xmbean>
      <attribute access="read-only" getMethod="getPolicyProxy">
        <description>The java.security.Policy implementation</description>
        <name>PolicyProxy</name>
        <type>java.security.Policy</type>
      </attribute>
      <attribute access="read-write" getMethod="getExternalPermissionTypes"

```

```

        setMethod="setExternalPermissionTypes">
        <description>The types of non-javax.security.jacc permissions that
            should be validated against this policy</description>
        <name>ExternalPermissionTypes</name>
        <type>[Ljava.lang.Class;</type>
    </attribute>
    <operation>
        <name>listContextPolicies</name>
        <return-type>java.lang.String</return-type>
    </operation>
</mbean>
<!-- Not used, just here to test that custom permissions don't break the
current behavior of javax.security.jacc.* permissions.
-->
<attribute name="ExternalPermissionTypes">org.jboss.security.srp.SRPPPermission</attribute>
</mbean>
<mbean code="org.jboss.security.jacc.SecurityService"
    name="jboss.security:service=JaccSecurityService"
    xmlns-dd=" " >
    <xmbean>
        <descriptors>
            <injection id="MBeanServerType" setMethod="setMBeanServer" />
            <injection id="ObjectNameType" setMethod="setObjectName" />
        </descriptors>
        <attribute access="read-write" getMethod="getPolicyName" setMethod="setPolicyName">
            <description>The policy provider MBean name</description>
            <name>PolicyName</name>
            <type>javax.management.ObjectName</type>
        </attribute>
        <attribute access="read-write" getMethod="getPolicyAttributeName"
            setMethod="setPolicyAttributeName">
            <description>The Policy attribute name on the PolicyName MBean</description>
            <name>PolicyAttributeName</name>
            <type>java.lang.String</type>
        </attribute>
        <operation>
            <name>start</name>
        </operation>
        <operation>
            <name>stop</name>
        </operation>
    </xmbean>
    <attribute name="PolicyName">jboss.security:service=JaccPolicyProvider</attribute>
    <attribute name="PolicyAttributeName">PolicyProxy</attribute>
</mbean>
</server>

```

Hibernate Integration

5.1. Hibernate Mapping Files

Persistent classes that are mapped using Hibernate `.hbm.xml` files are supported. The EJB3 Deployer will search the archive for any `.hbm.xml` files and add them to the definition of the underlying Hibernate SessionFactory. These `.hbm.xml` files can be virtually anywhere within the archive under any java package or directory.

Class Mappings defined in `.hbm.xml` files can be managed by EntityManagers just as annotated `@Entity` beans are. Also, you are allowed to have relationships between a `.hbm.xml` mapped class and an EJB3 entity. So, mixing/matching is allowed.

5.2. Injection Hibernate Session and SessionFactory

You can inject a `org.hibernate.Session` and `org.hibernate.SessionFactory` directly into your EJBs just as you can do with EntityManagers and EntityManagerFactory. The behavior of a Session is just the same as the behavior of an injected EntityManager. The application server controls the lifecycle of the Session so that you do not have to open, flush, or close the session. Extended persistence contexts also work with injected Hibernate Sessions.

```
import org.hibernate.Session;
import org.hibernate.SessionFactory;

@Stateful public class MyStatefulBean ... {
    @PersistenceContext(unitName="crm") Session session1;
    @PersistenceContext(unitName="crm2", type=EXTENDED) Session extendedpc;
    @PersistenceUnit(unitName="crm") SessionFactory factory;
}
```

5.3. Access to org.hibernate.Session

You can get access to the current underlying Hibernate Session by typecasting your reference to EntityManager.

```
@PersistenceContext EntityManager entityManager;
public void someMethod();
{
    org.jboss.ejb3.entity.HibernateSession hs = (HibernateSession)entityManager;
    org.hibernate.Session session = hs.getHibernateSession();
}
```

5.4. Access to org.hibernate.Query

You can get access to the current underlying Hibernate Query by typecasting your reference to a `org.hibernate.ejb.QueryImpl`.

```
@PersistenceContext EntityManager entityManager;
public void someMethod()
{
    javax.persistence.Query query = entityManager.createQuery(...);
    org.hibernate.ejb.QueryImpl hs = (QueryImpl)query;
    org.hibernate.Query hbQuery = hs.getHibernateQuery();
}
```


JBoss EJB 3.0 extensions

JBoss provides a few extensions to the EJB 3.0 spec. This chapter describes those features here.

6.1. @Service EJBs

An extension offered by JBoss EJB 3.0 is the notion of a `@org.jboss.annotation.ejb.Service` annotated bean. They are singleton beans and are not pooled, so only one instance of the bean exists in the server. They can have both `@Remote` and `@Local` interfaces so they can be accessed by java clients. When different clients look up the interfaces for `@Service` beans, all clients will work on the same instance of the bean on the server. When installing the bean it gets given a JMX `ObjectName` in the MBean server it runs on. The default is

```
jboss.j2ee:service=EJB3,name=<Fully qualified name of @Service bean>,type=service
```

You can override this default `ObjectName` by specifying the `objectName` attribute of the `@Service` annotation.

6.1.1. @org.jboss.annotation.ejb.Management interface

In addition to supporting `@Local` and `@Remote` interfaces, a `@Service` bean can also implement an interface annotated with `@Management`. This interface will wrap the bean as an MBean and install it in the JBoss MBean Server. The operations and attributes defined in the `@Management` interfaces become MBean operations and attributes for the installed MBean.. The underlying bean instance is the same as the one accessed via the `@Local` or `@Remote` interfaces.

6.1.2. Lifecycle Management and dependencies

Just as for normal MBeans running in JBoss, `@Service` lifecycle management beans support lifecycle management. This involves

- Lifecycle Methods.
- Dependencies.

6.1.2.1. Lifecycle Methods

Your `@Management` interface may contain methods with the following signatures

```
void create() throws Exception;  
void start() throws Exception;  
void stop();  
void destroy();
```

Each of these methods corresponds to when the MBean enters a particular state, so you can handle that. You do not need to include any of these methods, and you can pick and choose which ones you want to use. A descrip-

tion of the MBean states these methods correspond to:

- `create()` - called by the server when the service is created and all the services it depends upon have been created too. At this point the service (and all the services it depends on) is installed in the JMX server, but is not yet fully functional.
- `start()` - called by the server when the service is started and all the services it depends upon have been started too. At this point the service (and all the services it depends on) is fully functional..
- `stop()` - called by the server when the service is stopped. At this point the service (and all the services that depend on it) is no longer fully operational.
- `destroy()` - called by the server when the service is destroyed and removed from the MBean server. At this point the service (and all the services that depend on it) are destroyed.

6.1.2.2. Dependencies

We mentioned dependencies between MBeans in the previous section. You can specify what MBeans you depend on by using the `org.jboss.annotation.ejb.Depends` annotation. This will also work for "proper" EJBs, i.e. ones annotated with `@Stateful`, `@Stateless` and `@MessageDriven`. The `Depends` annotation-type takes an array of the String representation of the JMX ObjectNames of the service we depend on.

```
@Target({ElementType.TYPE, ElementType.METHOD, ElementType.FIELD})
@Retention(RetentionPolicy.RUNTIME)
public @interface Depends
{
    String[] value();
}
```

It can either be used on class level, in which case it simply means "I depend on these services". It can also be used on fields or setters in the bean class that take a type of a `@Management` annotated interface. In addition to the meaning of annotating at class level it will inject the `@Management` interface of the service we depend on.

6.1.3. Example

First we define a `@Management` interface

```
package org.jboss.tutorial.service.bean;

import org.jboss.annotation.ejb.Management;

@Management
public interface ServiceOneManagement
{
    void setAttribute(int attribute);
    int getAttribute();
    String sayHello();

    void create() throws Exception;
    void start() throws Exception;
    void stop();
    void destroy();
}
```

We also have a Remote interface

```
package org.jboss.tutorial.service.bean;

public interface ServiceOneRemote
{
    public void setAttribute(int attribute);
    public int getAttribute();
}
```

And finally we have a `@Service` annotated bean implementing our interfaces

```
package org.jboss.tutorial.service.bean;

import org.test.OtherServiceManagement;

import javax.ejb.Remote;
import org.jboss.annotation.ejb.Service;
import org.jboss.annotation.ejb.Depends;

@Service (objectName = "jboss:custom=Name")
@Remote(ServiceOneRemote.class)
@Depends ({ "jboss:service=someService" })
public class ServiceOne implements ServiceOneRemote, ServiceOneManagement
{
    int attribute;

    @Depends({ "jboss:j2ee:service=EJB3,name=org.test.OtherService,type=service" })
    OtherServiceManagement other;

    public void setAttribute(int attribute)
    {
        this.attribute = attribute;
    }

    public int getAttribute()
    {
        return this.attribute;
    }

    public String sayHello()
    {
        return "Hello from service One";
    }

    // Lifecycle methods
    public void create() throws Exception
    {
        System.out.println("ServiceOne - Creating");
    }

    public void start() throws Exception
    {
        System.out.println("ServiceOne - Starting");
    }

    public void stop()
    {
        System.out.println("ServiceOne - Stopping");
    }

    public void destroy()
    {
        System.out.println("ServiceOne - Destroying");
    }
}
```

This bean is accessible from remote clients via the `ServiceOneRemote` interface. The `ServiceOneManagement`

interface defines the operations to be made available via JMX. It specifies all the lifecycle methods so we will get notified when this bean is created, started, stopped and destroyed. Our bean uses a custom object name, so it will be installed in the MBean server under the name `jboss:custom=Name`. At bean level we have specified that it depends on a service with the name `jboss:service=someService`, so that service must be started before our bean can start. Also, we have specified that we depend on another JMX service called `jboss.j2ee:service=EJB3,name=org.test.OtherService,type=service` and that we want to inject an MBean Proxy implementing the `org.test.OtherServiceManagement` interface into the `other` field. For a more complete example, take a look at the tutorial.

6.2. Message Driven POJOs

Doco not complete yet. See tutorial for more information.

6.3. Asynchronous invocations

In JBoss EJB 3.0, you can convert a reference to a local or remote session bean into a reference that will invoke asynchronously on the EJB. This is called obtaining an asynchronous proxy or interface. asynchronous interface. Methods called on the asynchronous proxy will be executed asynchronously, and the results can be obtained later on. You obtain an asynchronous proxy by invoking on the `org.jboss.ejb3.asynchronous.Async.getAsynchronousProxy` static method. You pass

```
package org.jboss.ejb3.asynchronous;

public class Async
{
    public static <T> T getAsynchronousProxy(T ejbRef) {...}

    public static Future getFutureResult(Object asynchProxy)
    }
}
```

The asynchronous proxy created will implement the same local/remote interfaces as the original proxy. When you invoke on a method of the asynch proxy, the method will be invoked in the background. If the method has a return value, the value returned by the method will be null or zero depending on the return type. This will not be the true return value of the asynch method so you can just throw this away. To obtain the real result you must obtain `org.jboss.aspects.asynch.Future` object. This Future object works much the same way a `java.util.concurrent.Future` object does.

Let us look at an example. We have the following session bean.

```
package org.acme.test;

@javax.ejb.Remote
public interface Test
{
    int performHeavyOperation(int i);
    String echo(String s);
}
```

```
package org.acme.test;

@javax.ejb.Stateless
```

```
public class TestBean implements Test
{
    public int performHeavyOperation(int i)
    {
        try
        {
            //Simulate time-consuming operation
            Thread.sleep(10000);
            return i;
        }
        catch(Exception e)
        {
            throw new RuntimeException(e);
        }
    }

    public String echo(String s)
    {
        return s;
    }
}
```

As you can see there is nothing special about the session bean, now let's look at the client code. First we look up the remote interface:

```
InitialContext ctx = new InitialContext();
Test test = (Echo)ctx.lookup(org.acme.test.Test.getName());
```

We now have a reference to the bean's normal interface. Calls done on this interface will execute synchronously. The following call to `performHeavyOperation()` will block the client thread for 10 seconds.

```
int i = test.performHeavyOperation(1);
//i will be 1
```

Now to demonstrate the asynchronous functionality, we use the asynchronous proxy.

```
Test asynchTest = org.jboss.ejb3.asynchronous.Asynch.getAsynchronousProxy(test);
```

Calls made on the asynchronous interface will return 0 in the case of a simple return type or null in the case of an Object return type. We will see how to obtain the return value further down.

```
int j = asynchTest.performHeavyOperation(123);
//j will be 0
```

The call to `performHeavyOperation()` returns immediately, and our client thread is now free to do other stuff while the business method executes.

```
//You can do other stuff in client's thread
```

Now that we have finished doing things in our thread while the business method has been executing on the server, we obtain the `Future` which will hold the result of our asynchronous invocation.

```
Future future = org.jboss.ejb3.asynchronous.Asynch.getFutureResult(asynchTest);
```

It is important to note that you must call `getFutureResult()` to obtain the future for the last method call. If you call another method on the async proxy, then you will lose the previous `Future`. The asynchronous invocation might not have finished yet (in case the extra things we did in the client code took < 10 seconds), so we check for this here:

```
while (!future.isDone())
{
    Thread.sleep(100);
}
```

Now that the asynchronous invocation is done, we can obtain its return value from the `Future` object.

```
int ret = (String)future.get();
//ret will be 123
```

JBoss EJB 3.0 jboss.xml deployment descriptor

JBoss supports several vendor specific extensions EJB 3.0. The extensions are specified either as source code annotations or through the jboss.xml deployment descriptor. The chapter discusses the tags and options of the EJB 3.0 jboss.xml deployment descriptor.

7.1. Bean extensions

JBoss offers the capability to create additional Service and Consumer/Producer bean types.

7.1.1. Service

JBoss offers the capability to create services through the `@Service` annotation or through the jboss.xml deployment descriptor. They are singleton beans and are not pooled. Service beans can expose both local and remote interfaces so they can be accessed from java clients. When different clients access the interfaces for Service beans, all clients will be accessing the same instance on the server. Below is an example of a jboss.xml deployment descriptor that specifies a Service.

The implementing class is specified by the `ejb-class` tag. Local and remote interfaces are specified with the `local` and `remote` tags, respectively. JNDI bindings are specified through the `jndi-name` and the `local-jndi-name` tags.

A Service bean can also implement a management interface, which wraps the MBean and installs the Service as an MBean on the MBean server. The management interface is specified by the `management` tag. The management interface can support the standard lifecycle management of JBoss JMX.

When deployed, the bean is assigned a JMX ObjectName in the MBean server. The ObjectName can be explicitly set with the `object-name` tag. The default is: `jboss.j2ee:service=EJB3,name=<Fully qualified name of @Service bean>,type=service`.

Here is the example remote interface:

```
public interface ServiceSixRemote
{
    boolean getCalled();
    void setCalled(boolean called);
    void remoteMethod();
}
```

Here is the example local interface:

```
public interface ServiceSixLocal
{
    boolean getCalled();
    void setCalled(boolean called);
}
```

```
void localMethod();  
}
```

Here is the example management interface:

```
public interface ServiceSixManagement  
{  
    String jmxOperation(String s);  
    String[] jmxOperation(String[] s);  
    int getAttribute();  
    void setAttribute(int i);  
    int getSomeAttr();  
    void setSomeAttr(int i);  
    int getOtherAttr();  
    void setOtherAttr(int i);  
    void setWriteOnly(int i);  
    int getReadOnly();  
  
    void create() throws Exception;  
    void start() throws Exception;  
    void stop();  
    void destroy();  
}
```

Here is the implementing Service bean class:

```
public class ServiceSix implements ServiceSixLocal, ServiceSixRemote, ServiceSixManagement  
{  
    boolean called;  
  
    int localMethodCalls;  
    int remoteMethodCalls;  
    int jmxAttribute;  
    int someJmxAttribute;  
    int otherJmxAttribute;  
    int readWriteOnlyAttribute;  
  
    public boolean getCalled()  
    {  
        return called;  
    }  
  
    public void setCalled(boolean called)  
    {  
        this.called = called;  
    }  
  
    public void localMethod()  
    {  
        called = true;  
    }  
  
    public void remoteMethod()  
    {  
        called = true;  
    }  
  
    public String jmxOperation(String s)  
    {  
        return "x" + s + "x";  
    }  
  
    public String[] jmxOperation(String[] s)  
    {  

```



```
        return s;
    }

    public int getAttribute()
    {
        return jmxAttribute;
    }

    public void setAttribute(int i)
    {
        jmxAttribute = i;
    }

    public int getSomeAttr()
    {
        return someJmxAttribute;
    }

    public void setSomeAttr(int i)
    {
        someJmxAttribute = i;
    }

    public int getOtherAttr()
    {
        return otherJmxAttribute;
    }

    public void setOtherAttr(int i)
    {
        otherJmxAttribute = i;
    }

    public void setWriteOnly(int i)
    {
        readWriteOnlyAttribute = i;
    }

    public int getReadOnly()
    {
        return readWriteOnlyAttribute;
    }

    public void create() throws Exception
    {
    }

    public void start() throws Exception
    {
    }

    public void stop()
    {
    }

    public void destroy()
    {
    }
```

Here is the example deployment descriptor:

```
<jboss>
  <enterprise-beans>
    <service>
      <ejb-class>org.jboss.ejb3.test.service.ServiceSix</ejb-class>
      <local>org.jboss.ejb3.test.service.ServiceSixLocal</local>
      <remote>org.jboss.ejb3.test.service.ServiceSixRemote</remote>
```

```

        <management>org.jboss.ejb3.test.service.ServiceSixManagement</management>
        <jndi-name>serviceSix/remote</jndi-name>
        <local-jndi-name>serviceSix/local</local-jndi-name>
    </service>
</enterprise-beans>
</jboss>

```

7.1.2. Consumer/Producer

JBoss offers the capability to create message producers and consumers through the `@Producer` and `@Consumer` annotations or through the `jboss.xml` deployment descriptor. The Consumer class is specified using the `consumer` and `ejb-class` tags. The destination and destination type are specified in the `destination` and `destination-type` tags. The Producer interface is specified by the `producer` tag. Messages are created and sent to the configured destination when one of the Producer methods is called. A Consumer is configured to listen on a destination and when a message arrives, the current message is set in the implementing Consumer bean based on the `current-message` tag.

Here is the example for the remote and Producer interfaces:

```

public interface DeploymentDescriptorQueueTestRemote
{
    void method1(String msg, int num);

    void method2(String msg, float num);
}

```

Here is the example for the implementing Consumer/Producer bean class

```

public class DeploymentDescriptorQueueTestConsumer implements DeploymentDescriptorQueueTestRemote
{
    Message currentMessage;

    private Message setterMessage;

    void setMessage(Message msg)
    {
        setterMessage = msg;
    }

    public void method1(String msg, int num)
    {
        TestStatusBean.queueRan = "method1";
        TestStatusBean.fieldMessage = currentMessage != null;
        TestStatusBean.setterMessage = setterMessage != null;

        System.out.println("method1(" + msg + ", " + num + ")");
    }

    public void method2(String msg, float num)
    {
        TestStatusBean.queueRan = "method2";

        TestStatusBean.fieldMessage = currentMessage != null;
        TestStatusBean.setterMessage = setterMessage != null;

        System.out.println("method2(" + msg + ", " + num + ")");
    }
}

```

Here is an example code snippet for a Consumer/Producer client. When the `tester.method#` methods are called, the current message is set in the Consumer bean.

```
public void testDeploymentDescriptorQueue() throws Exception
{
    DeploymentDescriptorQueueTestRemote tester = (DeploymentDescriptorQueueTestRemote) getInitialContext().lookup("tester");
    ProducerManager manager = (ProducerManager) ((ProducerObject) tester).getProducerManager();
    manager.connect();
    try
    {
        tester.method1("testQueue", 5);

        tester.method2("testQueue", 5.5F);
    }
    finally
    {
        manager.close();
    }
}
```

Here is the example deployment descriptor:

```
<jboss>
  <enterprise-beans>
    <consumer>
      <ejb-name>DeploymentDescriptorQueueTestConsumer</ejb-name>
      <ejb-class>org.jboss.ejb3.test.consumer.DeploymentDescriptorQueueTestConsumer</ejb-class>
      <destination>queue/consumertest</destination>
      <destination-type>javax.jms.Queue</destination-type>
      <producer>org.jboss.ejb3.test.consumer.DeploymentDescriptorQueueTestRemote</producer>
      <current-message>
        <method>
          <method-name>currentMessage</method-name>
        </method>
        <method>
          <method-name>setMessage</method-name>
        </method>
      </current-message>
      <message-properties>
        <method>
          <method-name>method2</method-name>
        </method>
        <delivery>NonPersistent</delivery>
      </message-properties>
    </consumer>
  </enterprise-beans>
</jboss>
```

JBoss EJB 3.0 partial deployment descriptors

EJB 3.0 allows for partial deployment descriptors to augment or override the behavior of source code annotations. This chapter describes the use of partial deployment descriptors.

8.1. Overview

Beans in EJB 3.0 can be specified via source code annotations and/or a deployment descriptor. The deployment descriptor is used to augment or override the source code annotations. There are some limitations on which annotations may be overridden, however. The annotations that specify the bean itself (e.g. `@Stateless`, `@Stateful`, `@MessageDriven`, `@Service`, `@Consumer`) cannot be overridden. The EJB 3.0 `ejb-jar.xml` deployment descriptor DTD specifies the majority of tags as optional in order to support annotation augmentation and overrides. The deployment descriptor does not need to specify all of the required information, just that additional information to override or augment the source code annotations.

8.2. Examples

This section contains examples of complete and partial deployment descriptors for completely specifying or overriding specific behaviors of EJBs.

8.2.1. Complete deployment descriptor

The following `ejb-jar.xml` file contains a complete specification for a series of EJBs, including tags for security, transactions, resource injection, references, callbacks, callback listeners, interceptors, etc.

```
<ejb-jar>
  <description>jBoss test application </description>
  <display-name>Test</display-name>
  <enterprise-beans>
    <session>
      <ejb-name>Teller</ejb-name>
      <remote>org.jboss.ejb3.test.bank.Teller</remote>
      <ejb-class>org.jboss.ejb3.test.bank.TellerBean</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
      <ejb-ref>
        <ejb-ref-name>ejb/Bank</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <remote>org.jboss.ejb3.test.bank.Bank</remote>
        <ejb-link>Bank</ejb-link>
        <injection-target>bank</injection-target>
      </ejb-ref>
      <resource-ref>
        <res-ref-name>java:/TransactionManager</res-ref-name>
        <res-type>javax.transaction.TransactionManager</res-type>
        <res-auth>Container</res-auth>
        <res-sharing-scope>Shareable</res-sharing-scope>
        <injection-target>setTransactionManager</injection-target>
      </resource-ref>
    </session>
  </enterprise-beans>
</ejb-jar>
```

```

</resource-ref>
<resource-ref>
  <res-ref-name></res-ref-name>
  <res-type>javax.ejb.TimerService</res-type>
  <res-auth>Container</res-auth>
  <res-sharing-scope>Shareable</res-sharing-scope>
  <injection-target>ts</injection-target>
</resource-ref>
<security-identity>
  <run-as>
    <role-name>bankTeller</role-name>
  </run-as>
</security-identity>
</session>
<session>
  <ejb-name>Bank</ejb-name>
  <remote>org.jboss.ejb3.test.bank.Bank</remote>
  <ejb-class>org.jboss.ejb3.test.bank.BankBean</ejb-class>
  <session-type>Stateful</session-type>
  <transaction-type>Container</transaction-type>
  <env-entry>
    <env-entry-name>id</env-entry-name>
    <env-entry-type>java.lang.String</env-entry-type>
    <env-entry-value>5678</env-entry-value>
  </env-entry>
  <resource-ref>
    <res-ref-name>java:DefaultDS</res-ref-name>
    <res-type>javax.sql.DataSource</res-type>
    <res-auth>Container</res-auth>
    <res-sharing-scope>Shareable</res-sharing-scope>
    <injection-target>customerDb</injection-target>
  </resource-ref>
  <interceptor>org.jboss.ejb3.test.bank.FirstInterceptor</interceptor>
  <interceptor>org.jboss.ejb3.test.bank.SecondInterceptor</interceptor>
  <callback-listener>org.jboss.ejb3.test.bank.ExternalCallbackListener</callback-listener>
</session>
</enterprise-beans>
<assembly-descriptor>
  <callback>
    <annotation>PostConstruct</annotation>
    <method>
      <ejb-name>Teller</ejb-name>
      <method-name>postConstruct</method-name>
    </method>
  </callback>
  <remove-list>
    <method>
      <ejb-name>Bank</ejb-name>
      <method-name>remove</method-name>
    </method>
  </remove-list>
  <init-list>
    <method>
      <ejb-name>Bank</ejb-name>
      <method-name>init</method-name>
    </method>
  </init-list>
  <security-role>
    <role-name>bankCustomer</role-name>
  </security-role>
  <security-role>
    <role-name>bankTeller</role-name>
  </security-role>
  <method-permission>
    <role-name>bankCustomer</role-name>
    <method>
      <ejb-name>Teller</ejb-name>
      <method-name>greetChecked</method-name>
    </method>
  </method-permission>
  <method-permission>

```

```

    <unchecked/>
    <method>
      <ejb-name>Teller</ejb-name>
      <method-name>greetUnchecked</method-name>
    </method>
  </method-permission>
  <method-permission>
    <role-name>bankTeller</role-name>
    <method>
      <ejb-name>Bank</ejb-name>
      <method-name>getCustomerId</method-name>
    </method>
    <method>
      <ejb-name>Bank</ejb-name>
      <method-name>storeCustomerId</method-name>
    </method>
  </method-permission>
  <container-transaction>
    <method>
      <ejb-name>Teller</ejb-name>
      <method-name>greetWithNotSupportedTransaction</method-name>
    </method>
    <trans-attribute>NotSupported</trans-attribute>
  </container-transaction>
  <container-transaction>
    <method>
      <ejb-name>Teller</ejb-name>
      <method-name>greetWithRequiredTransaction</method-name>
      <method-params>
        <method-param>java.lang.String</method-param>
      </method-params>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
  <container-transaction>
    <method>
      <ejb-name>Bank</ejb-name>
      <method-name>*</method-name>
    </method>
    <trans-attribute>Required</trans-attribute>
  </container-transaction>
  <exclude-list>
    <method>
      <ejb-name>Teller</ejb-name>
      <method-name>excludedMethod</method-name>
    </method>
  </exclude-list>
</assembly-descriptor>
</ejb-jar>

```

8.2.2. Security

The following `ejb-jar.xml` file overrides any `@RolesAllowed`, `@PermitAll`, or `@DenyAll` source code annotations for the `bar` method of the `FooB` EJB and adds a `@RolesAllowed` annotation for the `allowed` role.

```

<ejb-jar>
  <assembly-descriptor>
    <method-permission>
      <role-name>allowed</role-name>
      <method>
        <ejb-name>FooB</ejb-name>
        <method-name>bar</method-name>
      </method>
    </method-permission>
  </assembly-descriptor>
</ejb-jar>

```

8.2.3. Transactions

The following `ejb-jar.xml` file overrides any `@TransactionAttribute` annotations for the `bar` method of the `FooA` EJB and adds a `@TransactionAttribute` annotation for `NOT SUPPORTED`.

```
<ejb-jar>
  <assembly-descriptor>
    <container-transaction>
      <method>
        <ejb-name>FooA</ejb-name>
        <method-name>bar</method-name>
      </method>
      <trans-attribute>NotSupported</trans-attribute>
    </container-transaction>
  </assembly-descriptor>
</ejb-jar>
```

8.2.4. References

The following `ejb-jar.xml` file creates a local jndi EJB reference and injects the EJB to the injection target of the `bank` member variable.

```
<ejb-jar>
  <enterprise-beans>
    <session>
      <ejb-name>Teller</ejb-name>
      <ejb-ref>
        <ejb-ref-name>ejb/Bank</ejb-ref-name>
        <ejb-ref-type>Session</ejb-ref-type>
        <remote>org.jboss.ejb3.test.bank.Bank</remote>
        <ejb-link>Bank</ejb-link>
        <injection-target>bank</injection-target>
      </ejb-ref>
    </session>
  </enterprise-beans>
</ejb-jar>
```

8.2.5. Callbacks

The following `ejb-jar.xml` file adds a `@PostConstruct` annotation to the `postConstruct` method of the `Teller` EJB.

```
<ejb-jar>
  <assembly-descriptor>
    <callback>
      <annotation>PostConstruct</annotation>
      <method>
        <ejb-name>Teller</ejb-name>
        <method-name>postConstruct</method-name>
      </method>
    </callback>
  </assembly-descriptor>
</ejb-jar>
```

JBoss EJB 3.0 MDB JCA Inflow

This chapter provides two examples on JCA 1.5 resources that allow JMS delivery of messages to EJB 3.0 Message Driven Beans. The first example uses SwiftMQ as the JCA XA resource. The second example illustrates the usage of non-standard (i.e. not `javax.jms.MessageListener`) messaging types.

9.1. SwiftMQ JCA XA Resource

SwiftMQ provide a JCA 1.5 resource that allows JMS message delivery to an MDB. The EJB 3.0 distribution provides a tutorial on building and deploying this example. The `@org.jboss.annotation.ejb.ResourceAdapter` annotation is used to designate the resource adapter you will be using. It takes an attribute that is the base name of the RAR deployment file name.

Here is the source listing for the MDB:

```
@MessageDriven(name="test/mdb", activationConfig =
{
    @ActivationConfigProperty(propertyName="messagingType", propertyValue="javax.jms.MessageListener"),
    @ActivationConfigProperty(propertyName="destinationType", propertyValue="javax.jms.Queue"),
    @ActivationConfigProperty(propertyName="Destination", propertyValue="testqueue"),
    @ActivationConfigProperty(propertyName="ConnectionFactoryName", propertyValue="ConnectionFactory"),
    @ActivationConfigProperty(propertyName="Transacted", propertyValue="true"),
    @ActivationConfigProperty(propertyName="Xa", propertyValue="true"),
    @ActivationConfigProperty(propertyName="DeliveryOption", propertyValue="B"),
    @ActivationConfigProperty(propertyName="SubscriptionDurability", propertyValue="Durable"),
    @ActivationConfigProperty(propertyName="MaxPoolSize", propertyValue="20"),
    @ActivationConfigProperty(propertyName="MaxMessages", propertyValue="1"),
})
@ResourceAdapter("swiftmq.rar")
public class AnnotatedTestMDBBean
    implements MessageListener
{
    public void onMessage(Message message)
    {
        System.out.println(message);
    }
}
```

9.2. Non-Standard Messaging Types

JBoss offers the capability use non-standard messaging types for Message Driven Beans. Standard MDBs implement the `javax.jms.MessageListener` interface. MDBs can alternatively implement a custom messaging interface. The EJB 3.0 distribution test suite provides a complete example for the usage of this bean.

Here is the source listing for a non-standard messaging type:

```
@MessageDriven(name = "TestMDB", activationConfig =
{
```



```
@ActivationConfigProperty(propertyName="messagingType", propertyValue="org.jboss.ejb3.test.jca.inflow")
@ActivationConfigProperty(propertyName="name", propertyValue="testInflow"),
@ActivationConfigProperty(propertyName="anInt", propertyValue="5"),
@ActivationConfigProperty(propertyName="anInteger", propertyValue="55"),
@ActivationConfigProperty(propertyName="localhost", propertyValue="127.0.0.1"),
@ActivationConfigProperty(propertyName="props", propertyValue="key1=value1,key2=value2,key3=value3")
})
@ResourceAdapter("jcainflow.rar")
@TransactionAttribute(TransactionAttributeType.REQUIRED)
public class TestMDBMessageListener implements TestMessageListener
{
    private static final Logger log = Logger.getLogger(TestMDBMessageListener.class);

    public void deliverMessage(TestMessage message)
    {
        message.acknowledge();
        log.info(message.toString());
    }
}
```

10

Transports

This chapter explains how remote clients communicate with the EJB 3 container and describes how to set up alternate transports. types. The transport used by JBoss EJB 3 is based on JBoss Remoting, see the JBoss Remoting documentation for more in-depth examples.

10.1. Default transport

By default JBoss EJB 3 uses a socket based invoker layer on port 3878. This is set up in `$JBOSS_HOME/server/<your-conf>/deploy/ejb3.deployer/META-INF/jboss-service.xml`

This is the out-of-the-box setup:

```
<mbean code="org.jboss.remoting.transport.Connector"
      xmbean-dd="org/jboss/remoting/transport/Connector.xml"
      name="jboss.remoting:type=Connector,name=DefaultEjb3Connector,handler=ejb3">
  <depends>jboss.aop:service=AspectDeployer</depends>
  <attribute name="InvokerLocator">socket://0.0.0.0:3873</attribute>
  <attribute name="Configuration">
    <handlers>
      <handler subsystem="AOP">org.jboss.aspects.remoting.AOPRemotingInvocationHandler</handler>
    </handlers>
  </attribute>
</mbean>
```

The URL given by the `InvokerLocator` attribute tells JBoss Remoting which protocol to use, and which IP address and port to listen on. In this case we are using `0.0.0.0` as the IP address, meaning we listen on all NIC's. The `socket://` protocol means we listen on a plain socket, and the port we use is `3873`

The `Configuration` attribute specifies that requests coming in via this socket should be handed over to the `AOP` subsystem, implemented by `org.jboss.aspects.remoting.AOPRemotingInvocationHandler` which is the entry point to the EJB 3 container.

10.2. Securing the transport

In some cases you may wish to use SSL as the protocol. In order to do this you first need to generate a keystore.

10.2.1. Generating the keystore and truststore

For SSL to work we need to create a public/private key pair, which will be stored in a keystore. Generate this using the `genkey` command that comes with the JDK.

```
$cd $JBOSS_HOME/server/default/conf/
$ keytool -genkey -alias ejb3-ssl -keypass opensource -keystore localhost.keystore
Enter keystore password: opensource
```

```

What is your first and last name?
[Unknown]:
What is the name of your organizational unit?
[Unknown]:
What is the name of your organization?
[Unknown]:
What is the name of your City or Locality?
[Unknown]:
What is the name of your State or Province?
[Unknown]:
What is the two-letter country code for this unit?
[Unknown]:
Is CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown correct?
[no]: yes

```

alias is the name ("ejb2-ssl") of the key pair within the keystore. keypass is the password ("opensource") for the keystore, and keystore specifies the location ("localhost.keystore") of the keystore to create/add to.

Since we have not signed our certificate through any certification authority, we also need to create a truststore for the client, explicitly saying that we trust the certificate we just created. The first step is to export the certificate using the JDK keytool:

```

$ keytool -export -alias ejb3-ssl -file mycert.cer -keystore localhost.keystore
Enter keystore password:  opensource
Certificate stored in file <mycert.cer>

```

Then we need to create the truststore if it does not exist and import the certificate into the truststore:

```

$ keytool -import -alias ejb3-ssl -file mycert.cer -keystore localhost.truststore
Enter keystore password:  opensource
Owner: CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
Issuer: CN=Unknown, OU=Unknown, O=Unknown, L=Unknown, ST=Unknown, C=Unknown
Serial number: 43bff927
Valid from: Sat Jan 07 18:23:51 CET 2006 until: Fri Apr 07 19:23:51 CEST 2006
Certificate fingerprints:
    MD5:  CF:DC:71:A8:F4:EA:8F:5A:E9:94:E3:E6:5B:A9:C8:F3
    SHA1: 0E:AD:F3:D6:41:5E:F6:84:9A:D1:54:3D:DE:A9:B2:01:28:F6:7C:26
Trust this certificate? [no]: yes
Certificate was added to keystore

```

10.2.2. Setting up the SSL transport

The simplest way to define an SSL transport is to define a new Remoting connector using the sslsocket protocol as follows. This transport will listen on port 3843:

```

<mbean code="org.jboss.remoting.transport.Connector"
  xmbean-dd="org/jboss/remoting/transport/Connector.xml"
  name="jboss.remoting:type=Connector,transport=socket3843,handler=ejb3">
  <depends>jboss.aop:service=AspectDeployer</depends>
  <attribute name="InvokerLocator">sslsocket://0.0.0.0:3843</attribute>
  <attribute name="Configuration">
    <handlers>
      <handler subsystem="AOP">org.jboss.aspects.remoting.AOPRemotingInvocationHandler</handler>
    </handlers>
  </attribute>
</mbean>

```

We need to tell JBoss Remoting where to find the keystore to be used for SSL and its password. This is done using the `javax.net.ssl.keyStore` and `javax.net.ssl.keyStorePassword=opensource` system properties when starting JBoss, as the following example shows:

```
$cd $JBoss_HOME/bin
$ run -Djavax.net.ssl.keyStore=../server/default/conf/localhost.keystore -Djavax.net.ssl.keyStorePassword=opensource
```

10.2.3. Configuring your beans to use the SSL transport

By default all the beans will use the default connector on `socket://0.0.0.0:3873`. By using the `@org.jboss.annotation.ejb.RemoteBinding` annotation we can have the bean invocable via SSL.

```
@RemoteBinding(clientBindUrl="sslsocket://0.0.0.0:3843", jndiBinding="StatefulSSL"),
@Remote(BusinessInterface.class)
public class StatefulBean implements BusinessInterface
{
    ...
}
```

This bean will be bound under the JNDI name `StatefulSSL` and the proxy implementing the remote interface returned to the client will communicate with the server via SSL.

You can also enable different types of communication for your beans

```
@RemoteBindings({
    @RemoteBinding(clientBindUrl="sslsocket://0.0.0.0:3843", jndiBinding="StatefulSSL"),
    @RemoteBinding(jndiBinding="StatefulNormal")
})
@Remote(BusinessInterface.class)
public class StatefulBean implements BusinessInterface
{
    ...
}
```

Now if you look up `StatefulNormal` the returned proxy implementing the remote interface will communicate with the server via the normal unencrypted socket protocol, and if we look up `StatefulSSL` the returned proxy implementing the remote interface will communicate with the server via SSL.

10.2.4. Setting up the client to use the truststore

If not using a certificate signed by a certificate authorization authority, you need to point the client to the truststore using the `javax.net.ssl.trustStore` system property and specify the password using the `javax.net.ssl.trustStorePassword` system property:

```
java -Djavax.net.ssl.trustStore=${resources}/test/ssl/localhost.truststore -Djavax.net.ssl.trustStorePassword=opensource
```